University of London
University College London
Department of Computer Science

Evaluating Architectural Stability with Real Options Theory

Rami K. Bahsoon

A thesis submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Faculty of Engineering Sciences of the
University of London

October 2005

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

Evolution is a key problem in software engineering and exacts huge costs. Industrial evidence even hints that companies spend more resources on maintaining and evolving their software than on the initial development. In managing the change and guiding evolution, considerable emphasis is placed on the architecture of the software system as a key artifact involved. One of the major indicators of the success (failure) of software evolution is the extent to which the software system can endure changes in requirements, while leaving the architecture of the software system intact. We refer to the presence of this "intuitive" phenomenon as architectural stability.

We highlight the requirements for evaluating architectural stability. We pursue an economics-driven software engineering approach to address these requirements. We view evolving software as a value-seeking activity: software evolution is as a process in which software is undergoing a change (an incremental) and seeking value. The value is attributed to the *flexibility* of an architecture in enduring likely changes in requirements. To value flexibility, we contribute to a novel model that builds on an analogy with real options theory. The model examines some likely changes in requirements and values the extent to which the architecture is flexible to endure these changes. The model views an investment in an architecture as an upfront investment plus "continual" increments of future investments in likely changes in requirements. The objective is to provide insights into architectural stability and investment decisions related to the evolution of software architectures.

We support the model with a three-phase method for evaluating architectural stability. The method provides guidelines on eliciting the likely changes in requirements and relating architectural decisions to value. The problem of valuing flexibility of an architecture to change requires a comprehensive solution that incorporates multiple valuation techniques, some with subjective estimates, and others based on market data, when available. To introduce discipline into this setting and capture the value from different perspectives, the method outlines a *valuation points of view* framework as a solution. The framework is flexible enough to account for the economic ramifications of the change on both structural (e.g., maintainability) and behavioral (e.g., throughput) qualities of an architecture and on relevant business goals (e.g., new market products).

We report on our experience in using the model and its supporting method with two case studies. In the first case, we show how the model and its supporting method can be used to assess the worthiness of re-engineering a "more" stable architecture in face of likely changes in future requirements. We take refactoring as an example of re-engineering. In the second case, we show how the model and its supporting method can inform the selection of a "more" stable middleware-induced software architecture in the face of future changes in non-functional requirements.

We critically discuss and reflect on the strengths and the limitations of our contribution. We conclude by highlighting some open questions that could stimulate future research in architectural stability, relating requirements to software architectures, and architectural economics.

Acknowledgements

With a debt of gratitude, which cannot be adequately expressed in words, I thank my supervisor Prof Wolfgang Emmerich for his advice, guidance, and endless support during my research. His practical and sharp vision in research has not only been invaluable for my work on this thesis but also for developing my taste in research and my development as a researcher. For the last four years, engaging in any discussion with Wolfgang has been an enjoyable practical lesson, where professionalism, devotion in duty, guaranteed honesty (black or white- never gray), gentleness, assured care, and a feel of protection are the masters. Thanks for being a friend and an excellent listener for many awkward matters (jobs, visas, etc.) and for the healthy student-focused environment, you have provided. I have been very fortunate to work with you Wolfgang. At the least, I promise to be loyal to all your teachings and lessons in professionalism.

I extend my sincere gratitude to my second supervisor Prof Anthony Finkelstein for his guidance, insightful suggestions, and encouragements during various stages of my PhD. I am indebted to Anthony for the many stimulating and exciting discussions, which have trained me in research; for patiently listening to many bizarre thoughts (Life-Oriented RE- it rings a bell?); and for his paternal spirit, which eased my early adaptation periods in the group. Thanks for your big heart.

I am deeply indebted to Prof Angela Sasse for all the unforgettable generous support during periods of the PhD process. I am also indebted to Prof David Rosenblum for all his invaluable support, gentleness, and his insights during the MPhil/PhD Transfer viva. Thanks to Dr Graham Roberts for all his internal reviews. Truthful thanks to Dr Cecilia Mascolo for the caring and harmonious spirit she spreads in the group. Sincere thanks to Dr Licia Capra for her friendship and all her sharp advices from day one (many to count and many to come, Licia). James Skene and Andy Dingwall Smith, I thank you for the impressive commitment in reading some chapters of the thesis.

The work has greatly benefited from discussion with many fellows at the Economics-Driven Software Engineering workshops for two consecutive meetings. In particular, I am indebted to the discussions and guidance of Prof Kevin Sullivan, Dr Hakan Erdogmus, Prof Eline Stroulia, and Vahe Poladian. I am also thankful to Prof David Noktin for his spirit as a great educator and for the invaluable input during the ICSE Doctoral Symposium. The generous support of Harry Sneed at ICSM and Dr Tony Wicks at Searchspace are unforgettable. The pleasant discussions and encouragements of Prof Bashar Nuseibeh have shown that Requirements Engineering has not just offered an influential researcher, but also an influential human. Thanks for the generous support of Dr Paul Brebner. The early but advanced research teachings of Dr Nashat Mansour and his continuous encouragement and support are significant - conveying a word of thanks continues to be little. My sincere gratitude to Dr Ramzi Haraty for all his teachings. Thanks to Prof Ian Nabney at Aston U. for the flexibility he provided to complete this thesis.

My years at UCL were very enjoyable, thanks for the friendship of: Christian(for all the unforgettable help), Carina(for all the stimulating discussions), Genna(I should have predicted the "implied scenarios" for all the enjoyable distractions), Mirco(for the Lebanese Italian social epidemic gym gossips), Stef(Stefanouli?- A brilliant context-aware Arabic learner), Toresten(for the political discussions), Leticia(for being my ambassador to Searchspace), Vlad(for the wisdom when "sensing" your jokes), Gacomo(for the unforgettable Dublin discussion), Luca, Anti, Clare, Nima, Danielle, Daniel, Panu, Vitto, Chiara, Ben, Costin, Clovis, Bruno, Evan.... A word of thanks, loyalty, and respect to those I have forgot to mention.

I express my sincere gratitude to my examiners Prof Kevin Ryan and Prof Rachel Harrison. It is a big honor to have you as examiners. Thanks for your kindness and your insightful thoughts.

Last but not least, I am deeply indebted to my family for their generous support and the financial sacrifice for making this possible; it was uneasy trip for either. Endless thanks to my father and mother for all the feelings, which I cannot really express, to my sister Rachel, and my brother Rabi for their endless emotional support. The least I can express is a modest prayer for my continuous loyalty and a hope to continue to be your tireless and obedient servant forever long.

To my parents with loyalty and love...

"I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you can not measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of Science, whatever the matter may be"

Lord Kelvin

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Definition

Software requirements, whether functional or non-functional, are generally volatile; they are likely to change and evolve over time. The change is inevitable as it reflects changes in stakeholders' needs and the environment in which the software system works. *Software architecture* is the earliest design artifact, which realizes the requirements of the software system. It is the manifestation of the earliest design decisions, which comprise the architectural structure (i.e., components and interfaces), the architectural topology (i.e., the architectural style), the architectural infrastructure (e.g., the middleware), the relationship among them, and their relationship to the other software artifacts (e.g., low-level design, testing etc.). One of the major implications of a software architecture is to render particular kinds of changes easy or difficult, thus constraining the software's evolution possibilities [Jazayeri, 2002]. A change may "break" the software architecture necessitating changes to the architectural structure (e.g., changes to components and interfaces), architectural topology, or even changes to the underlying architectural infrastructure. It may be expensive and difficult to change the architecture as requirements evolve [Finkelstein, 2000]. Conversely, failing to accommodate the change leads ultimately to the degradation of the usefulness of the system. Hence, there is a pressing need for flexible software architectures that tend to be *stable* as the requirements evolve. By a stable architecture, we mean the extent to which a

software system can endure changes in requirements, while leaving the architecture of the software system intact. We refer to the presence of this "intuitive" phenomenon as *architectural stability*.

Developing architectures which are (a) *stable* in the presence of change and (b) *flexible enough* to be customized and adapted to the changing requirements is one of the key challenges in software engineering [Garlan, 2000]. Ongoing research on relating requirements to software architectures has considered the architectural stability problem as an open research challenge and *difficult* to handle [van Lamsweerde, 2000; Nuseibeh, 2001]. In particular, van Lamsweerde [2000] acknowledges that "the conflict between requirements volatility and architectural stability is a difficult one to handle". Nuseibeh [2001] notes that many architectural stability related questions are difficult and remain unanswered. For example, what software architectures (or architectural styles) are stable in the presence of the changing requirements, and how do we select them? What kinds of changes are systems likely to experience in their lifetime, and how do we manage requirements and architectures (and their development processes) in order to manage the impact of these changes?

Meanwhile, evolution is still a key problem in software engineering and exacts huge costs [Jazayeri, 2002; Lehman et al., 2000]. Empirical evidence even hints that companies spend more resources on maintaining and evolving their software than on the initial development [Boehm and Sullivan, 2000; Jazayeri, 2002; Bennet and Rajlich, 2000; FEAST 1-2]. In managing the change and guiding evolution, considerable emphasis is placed on the architecture of the software system as the key artifact involved [Garlan, 2000; Jazayeri, 2002]. Cook, Ji, and Harrison note that "*In many software systems, the architecture is the level that has the greatest inertia when external circumstances change and consequently incurs the highest maintenance costs when evolution becomes unavoidable*" [Cook et al., 2001]. An established route to manage the change and guide evolution is a universal "design for change" philosophy, where the architecture is conceived and developed such that evolution is possible [Parnas, 1979]. Parnas's notion of the "design for change" is based on the recognition that much of the total lifecycle cost of a system is expended in the change and incurred in evolution. A system that is not designed for evolution will incur tremendous costs,

which are dispropionate to the benefits [Lientz and Swanson, 1980]. For a system to create value, the cost of a change increment should be proportional to the benefits delivered [Parnas, 1972]. "Design for change" is thus promoted as a value-maximizing strategy provided one could anticipate changes [Boehm and Sullivan, 2000]. The "Design for change" philosophy is believed to be a useful heuristic for developing flexible architectures that tend to be stable as requirements evolve. However, there is a general lack of adequate models and methods, which connect this technical engineering philosophy to value creation under given circumstances [Boehm and Sullivan, 2000]:

From an economic perspective, the change in requirements is a source of uncertainty that confronts an architecture during the evolution of the software system. The change places the investment in a particular architecture at risk. Conversely, designing for change incurs upfront costs and may not render future benefits. The benefits are uncertain, for the demand and the nature of the future changes are uncertain. The worthiness of designing or re-engineering an architecture for change involves a tradeoff between the upfront cost of enabling the change and the future value added by the architecture, if the change materializes. The value added, as a result of enabling the change on a given architecture, is a powerful heuristic which can provide a basis for analyzing: (i) the worthiness of designing for change, (ii) the worthiness of re-engineering the architecture, (iii) the retiring and replacement decisions of the architecture or its associated design artifacts, (iv) the decisions of selecting an architecture, architectural style, middleware, and/or design with desired stability requirements, and/or (v) the success (failure) of evolution.

Therefore, to cope with uncertainties, incomplete knowledge in an evolutionary context, and to mitigate risks in the investment, there is a critical need for *evaluating architectural stability*. Evaluating architectural stability aims at assessing the extent to which the system of a given architecture is evolvable, while leaving the architecture and its associated design decisions unchanged as the requirements change. The evaluation shall address the economic interplay between designing flexible architectures, evolving requirements, impact of the requirements change on the architecture, and their long-term cost and value implications. Such interplay is

critical for proactively understanding the architectural stability problem and many of its associated research questions, from an economics-driven software engineering perspective [EDSER 1-7, 1999-2005]. The evaluation has the promise to answer the following challenging key question: How much is "buying" flexibility to facilitate future changes and support the development (evolution) of potentially stable architectures worth?

The research questions being addressed in this thesis include the following [Bahsoon, 2003]: How can we systematically evaluate the stability of software architectures in the face of the changing requirements, taking an economics-driven approach? What are the requirements for such evaluation and how can we address these requirements? What are the implications of the pursued approach on some architecture-centric cases, with essential or desirable stability requirements? Subsequent Sections and Chapters develop these questions.

1.2 The Research Perspective

Sullivan et al. [1997] note that the important book of Shaw and Garlan on software architecture begins, *"As the size and complexity of software systems increase, the design and specification of overall system structure become more significant issues than the choice of algorithms and data structures..."* [Shaw and Garlan, 1996]. Sullivan et al. [1997] add, *"This statement is true, without a doubt. The problem in the field is that no serious attempt is made to characterize the link between structural decisions and value added"*. That is, the traditional focus of software architecture is more on structural and technical perfection than on value added. In addressing the architectural stability problem, our perspective aims at providing a compromise through linking structural decisions to value creation.

In particular, the thesis adopts an economics-driven software engineering perspective [EDSER 1-7, 1999-2005] to evaluate the stability of software architectures in the face of changing requirements. Traditionally, engineering software has been primarily a technical endeavor with minimal attention given to its economic context

[Boehm and Sullivan, 2000]. Design and implementation methods are proposed based on technical merits without making adequate links to the economic considerations. This is in stark contrast to the reality of software engineering. Engineering seeks to create value relative to the resources invested. Regardless of how we define "value", engineering software is essentially an irreversible capital investment [EDSER 1-7, 1999-2005]. Developing and evolving software, thus, should add value to the enterprise, just as any other capital expenditure. As such, the costs of developing and evolving software should not outweigh the returns from the product to achieve a net benefit.

In this perspective, the thesis adopts the view that software design and engineering activity is one of investing valuable resources under uncertainty with the goal of maximizing the value added [Baldwin and Clark, 1999; Sullivan 1996; EDSER 1-7, 1999-2005]. This view approximates to much industrial practice. In particular, the thesis views evolving software as a value-seeking and value-maximizing activity: software evolution is a process in which software is undergoing an incremental change and seeking value [Bahsoon and Emmerich, 2004a]. The thesis attributes the added value to the *flexibility* of the architecture in enduring changes in requirements. Means for achieving flexibility are typical architectural mechanisms or strategies that are built-in or adapted into the architecture with the objective of facilitating evolution and future growth. This could be in response to changes in functional (e.g., changes in features) or non-functional requirements (e.g., changes in scalability demands). For example, consider functionality that is likely to change and evolve over time: "componentizing" the functionality and hiding it behind negotiable and configurable interfaces is a simple example of such a mechanism. As we are assuming that the added value is attributed to flexibility, arriving at a "more" stable software architecture requires finding an architecture which maximizes the yield in the embedded or the adapted flexibility in an architecture relative to the likely changing requirements [Bahsoon and Emmerich, 2004a; Bahsoon and Emmerich 2000b]. Optimally, a stable architecture is an architecture that shall add value to the enterprise and the system as the requirements evolve. By valuing the flexibility of an architecture to change, we aim at providing the architect/analyst with a useful tool for reasoning about a crucial but previously intangible source of value. This value

can be then used for deriving "insights" into architectural stability and investment decisions related to evolving software.

1.3 The Research Objectives

The goal of the thesis is to develop a framework for systematically evaluating the stability of software architectures in the face of changes in requirements, taking an economics-driven approach. By taking an economics-driven approach for evaluating architectural stability, we intend to proactively assess the complexity and the economic ramifications of the likely critical changes in requirements and their impact on the software architecture. The evaluation aims at understanding (i) the tradeoff between the upfront cost of enabling a change on the architecture of the software system and the long-term future benefits as a result; (ii) the trade-off between the architectural "intactness" and the cost-effectiveness of amending the architecture to accommodate a change; (iii) the cost and the value implications of evolving the requirements of the architecture; (iv) the economics of flexibility, inflexibility, and over-flexibility of the architecture relative to a change; and/or (vi) the cost-effectiveness of the technical design and reengineering decisions for a change.

The framework aims at providing a basis for analyses supporting many *architecture-centric approaches to evolution*, with desirable or essential stability requirements. By architecture-centric approaches to evolution, we refer to approaches, which pursue the software architecture as the appropriate level of abstraction for reasoning about, managing and guiding the evolution of complex software systems, and "synchronizing" the software requirements with its detailed design and implementation. A distinctive feature of these approaches is that they explicitly account for the non-functional requirements, the so-called quality attributes. As the quality attributes comprise the most substantial properties of the system, the evolution of such properties can be best reasoned about and managed at the architectural level. For example, the current trend is to build distributed systems architectures with middleware technologies such as Java 2 Enterprise Edition (J2EE) and the Common Object Request Broker Architecture (CORBA), resulting in the so-

called middleware-induced architectures [Di Nitto and Rosenblum, 1999]. Middleware-induced architectures follow an architectural-centric approach to evolution, as the emphasis is placed on the induced architecture for simplifying the construction of distributed systems by providing high-level primitives, which shield the application engineers from the distribution complexities, managing systems resources, and implementing low-level details, such as concurrency control, transaction management, and network communication. These primitives are often responsible for realizing many of the non-functional requirements (e.g., scalability, fault tolerance, etc.) in the architecture of the system induced and facilitating their evolution over time. Another example is from product-line architectures. Product-lines, a family of products sharing the same architecture, inherently require domain-specific variation and evolution of various products. Due to the higher level of interdependency between the various software artifacts in a product-line, software evolution is too complex to be dealt with at the code level. As the focus is on the architecture for "easing" and guiding evolution, architecture-centric approaches to evolution place considerable emphasis on the *flexibility* of the architecture in responding to change. In this context, the framework intends to answer the following key question: how much is it worth "buying" flexibility to facilitate future changes and support the development (evolution) of potentially stable architectures?

The benefit of this work is that it provides the analyst/architect with "insights" into architectural stability and investment decisions related to the evolution of software architectures. The objective is to assist the analyst/architect in strategic *"what if"* analyses involving: valuing the long-term investment in a particular architecture; analyzing the trade-offs between two or more candidate software architectures for the long-term value; analyzing the strategic position of the enterprise- if the enterprise is highly centered on the software architecture (as is the case in web-based companies); valuing the worthiness of designing or reengineering for the change; and valuing the flexibility of the architecture and its associated artifacts relative to the change. The intellectual framework is most critical; it demonstrates that with *value-based* reasoning we can improve our ability to evaluate for architectural stability and develop software systems that need to adapt to the inevitable evolving requirements.

1.4 Assumptions

The major assumptions underlying the thesis are as follows:

- Changes in requirements could be predictable in advance. Chapter 5 of the thesis provides guidelines for eliciting likely changes in requirements that are critical for evaluating architectural stability.
- The cost of re-engineering or re-architecting an architecture for the change can be predicted a long time in advance using a similar development or an evolution experience. However, the prediction needs not be accurate as the framework we propose provides treatment to the uncertainty of the prediction.
- Adapting flexibility into the architecture of the software system is often a costly option: For example, flexibility often come with a price (e.g., through the provision of primitives for facilitating the change). Furthermore, the adapted flexibility might be underutilized to reveal a net benefit upon exercising the change.
- We look at systems that are intended to evolve. In Lehman's concept [FEAST 1-2], there are two types of systems: these are E-type systems and S-type systems. E-Type systems that are embedded in real world applications and are used by humans for everyday business functions. Examples might be customer service, order entry, payroll, operating systems, databases engines. S-Type systems are executable models of a formal specification. The success of this software is judged by how well it meets the specification. For E-Type systems the "real world" is dynamic and ever changing. As the real world changes the specification changes and the E-Type systems adapt to these changes. Hence, E-Type systems are evolvable. For S-Type systems the specification becomes invalid in the presence of change. In Lehman's terminology, we look at E-type systems.

1.5 Thesis Contribution

The Contribution in Brief

This thesis advances the understanding of the architectural stability problem from an economics-driven software engineering perspective [EDSER 1-7, 1999-2005]. The merits of such a contribution can not be overstated: revealing a new practical perspective in tackling an unaddressed problem; stimulating; and possibly motivating future research in architectural stability and related problems. Accordingly, this thesis should be regarded as a culmination of four years of independent "make a way" challenge into the concept and the problem, in the absence of very closely related research. The thesis makes the following specific contributions:

- Surveys research work on architecture evaluation and discusses their limitations in addressing architectural evaluation for stability.
- Highlights the requirements for evaluating architectural stability in the face of changing requirements from an economics-driven perspective.
- Describes a novel approach and devises a real-options based model, referred to as ArchOptions, for valuing the flexibility of an architecture to change. The model builds on a sound theory in financial engineering to provide insights into architectural stability and investment decisions related to the evolution of software architectures.
- Complements the model with a three-phase method for conducting an architectural evaluation for stability. The problem of valuing flexibility of an architecture to change requires a comprehensive solution that incorporates multiple valuation techniques, some with subjective estimates, and others based on market data, when available. To introduce discipline into this setting and capture the value from different perspectives, the method outlines a *valuation points of view* framework as a solution. The framework addresses the problem that valuing the flexibility of an architecture to likely changes in requirements is a multi-perspectives valuation problem. The framework is flexible enough to account for the economic ramifications of the change on the structural (e.g.,

maintainability) and behavioral (e.g., throughput) qualities of an architecture and the associated business goals.

- Applies the approach to two architecture-centric evolution case studies, with desirable stability requirements. These applications demonstrate novelty in the use of real options theory in software engineering and draw some preliminary observations, lessons, and insights that could stimulate future research in the area of relating requirements to software architectures. The applications also aim at verifying the thesis-related claims (outlined in the next Subsection).
- Highlights some open questions that could stimulate future research in architectural stability, relating software requirements and architectures, and architectural economics.

The Thesis "Storyline"

A survey [Bahsoon and Emmerich, 2003a] of architectural evaluation methods indicates that current approaches to architectural evaluation focus explicitly on construction and only implicitly, if at all, on the phenomenon of software "evolution". Despite their concern with "change", these methods do not address stability. When these methods address qualities like modifiability, they do not predict and measure the capability of the architecture to withstand change. According to Cook, Ji, and Harrison, the provision of such measure is important, because, for example, "*it assists the objective assessment of the lifetime costs and benefits of evolving software, and the identification of legacy situations, where a system or component is indispensable but can no longer be evolved to meet changing needs at economic cost*" [Cook et al., 2001]. Moreover, existing methods ignore any economic considerations and are driven by ways that are not optimal for long-term value creation. Factors such as flexibility often have impact on value creation [Boehm and Sullivan, 2000].

To bridge the gap, this thesis proposes an economics-driven approach for evaluating architectural stability in face of changing requirements [Bahsoon, 2003]. It is assumed that the software architecture's goal is to facilitate the system's evolution. Software evolution is viewed as a process in which a software system is undergoing a change

incrementally and seeking a value. The thesis highlights the requirements for evaluating architectural stability from an economics-driven software engineering perspective [EDSER 1-7, 1999-2005; Boehm and Sullivan 2000]. The thesis then claims that using strategic value-based reasoning we can address these requirements. In particular, the thesis argues that *real options theory* [Myers 1977; Myers 1987] is suited to assist in the evaluation. However, this begs the question: Why real options theory? Real options theory argues that *flexibility under uncertainty* can be viewed as values in the form of real options [Schwartz and Trigeorgis, 2000]; the value of these options lies in the enhanced flexibility to cope with uncertainty. This perspective is appealing to the architectural stability problem: unfortunately, future changes in software requirements are uncertain, as the demand for change, its nature, and its likelihood are often uncertain. Hence, change is the likely source of uncertainty that confronts the architecture during its lifetime. In the face of uncertainty, there is a pressing need for architectures, which are flexible *enough* to cope with change. This gives the need to value the flexibility of the architecture in the face of change. This value can then be used as a metric for predicting architectural stability [Bahsoon and Emmerich, 2004a, Bahsoon and Emmerich, 2004b, Bahsoon and Emmerich, 2003b]. The importance of the idea cannot be overemphasized: it gives the architect an ability to reason about a crucial but previously intangible source of value and to use it in the evaluation of architectural stability.

To value the flexibility of an architecture in the face of changing requirements, the thesis contributes to a novel model that exploits Black and Scholes (Nobel Prize winning) financial options theory [Black and Scholes, 1973]. The model is referred to as ArchOptions [Bahsoon et al., 2005, Bahsoon and Emmerich, 2004a, Bahsoon and Emmerich, 2004b, Bahsoon and Emmerich, 2003b]. In ArchOptions, investment opportunity in an architecture amounts to an upfront investment for developing the system of a given architecture plus "continuous" future investments for evolving the software in response to likely future changes in requirements. Briefly, ArchOptions examines critical *likely* changes in requirements and values the extent to which the architecture is flexible enough to withstand these changes. ArchOptions draws on a simple and intuitive analogy with Black and Scholes [1973] for valuing this flexibility. ArchOptions assumes that the architecture is the appropriate level of abstraction at which to reason about and analyze the evolution value, costs, and

investment opportunities. The model builds on a sound theory in financial engineering to provide insights into architectural stability, investment decisions related to the evolution of software architectures, and a basis for analyses for many architecture-centric evolution problems. The thesis describes how we have derived the ArchOptions model: the assumptions and the analogy made, its formulation, its sensitivity, and report on its possible interpretations and uses.

The thesis complements the model with a three-phase method for conducting an architectural evaluation for stability. The method provides guidelines on eliciting the likely changes in requirements; it pursues *scenarios* as a possible solution to describe the *likely* future changes in requirements that are critical to the evaluation. To trace the likely future change in requirements to the architecture, goals are extracted from scenarios [Anton, 1997] and then refined (e.g., [Dardenne and van Lamsweerde, 1993]) using guidance on how they could operationalized by the architecture. The objective is to trace the change and quantify the flexibility of the architecture in withstanding the scenario. The valuation using ArchOptions requires a comprehensive solution that incorporates multiple valuation techniques, some with subjective estimates, and others based on market data, when available. The problem associated with how to guide the estimation in this setting, we term as a *multiple perspectives valuation problem*. To introduce discipline into this setting and capture the value from different perspectives, the method suggests valuation points of view (i.e., market or subjective estimates) as a solution. The framework is comprehensive enough to account for the economic ramifications of the change, its global impact on the architecture, and on other architectural qualities. The solution aims to promote flexibility through incorporating both subjective estimates and/or explicit market value, when available.

The thesis uses case studies to empirically evaluate ArchOptions and explore its fitness in addressing two architecture-centric evolution cases, with desired stability requirements. In the first case, we apply ArchOptions to value the payoff of refactoring [Bahsoon and Emmerich, 2004a; Bahsoon and Emmerich, 2004b]. The application demonstrates how ArchOptions can be used to value the worthiness of reengineering for better support to likely future changes in requirements. In the

second case, we apply ArchOptions to inform the selection of stable middleware-induced software architecture in face of likely critical changes in non-functional requirements [Bahsoon et al., 2005]. In this case, we argue that the choice of a stable distributed software architecture has to be guided by the choice of the middleware and its flexibility in responding to future changes in non-functional requirements. We draw on a case study that adequately represents a medium-size component-based distributed architecture: we report on how a likely future change in scalability, as representative critical change in non-functional requirements, could impact the architectural structure of two versions, each induced with a distinct middleware, one with CORBA and the other with J2EE. We show how we can apply ArchOptions to value the flexibility of the induced-architectures and to guide the selection. Research wise, addressing these problems has resulted in novel applications of real options theory in valuing the payoff of refactoring and in informing the selection of middleware-induced software architectures. On the discipline level, the application of ArchOptions to the above cases has provided some preliminary observations, lessons, and insights that could stimulate future research in the area of relating requirements to software architectures. Consequently, these observations aim at advancing our understanding of the architectural stability problem, when addressed from an economics-driven software engineering perspective.

The thesis concludes by highlighting some open questions that could stimulate future research in architectural stability, relating requirements to software architectures, and architectural economics.

Thesis-Related Publications

The work presented in this thesis is based on and extends several papers that have been published in the last three years [Bahsoon et al., 2005, Bahsoon and Emmerich 2004a; Bahsoon and Emmerich 2004b; Bahsoon 2003; Bahsoon and Emmerich 2003a; Bahsoon and Emmerich 2003b]. This thesis should be regarded as the definitive account of the work.

1.6 Thesis Outline

In **Chapter 2**, we survey seminal work on software architecture evaluation methods and identify their limitations in addressing stability and evolution. We document research motivation and perspectives on architectural stability. We differentiate between two types of approaches for evaluating architectural stability; these are retrospective or predictive.

In **Chapter 3**, we highlight the requirements for evaluating architectural stability. We motivate the need for an economics-driven approach to address these requirements.

In **Chapter 4**, we pursue an economics-driven approach to address the requirements highlighted in Chapter 3. We motivate the use of real options theory as a solution. We devise a real option model, referred to as ArchOptions, to systematically evaluate architectural stability. We describe the analogy that ArchOptions make with real options theory. We report on ArchOptions formulation, its possible interpretation, its sensitivity, and its possible uses. We discuss valuation issues and assumptions. We provide an overview of closely related work on the use of real options is software design and engineering.

In **Chapter 5**, we support the model with a three-phase method for evaluating architectural stability. We provide guidelines on applying ArchOptions and discuss practical ways for estimating the model parameters.

In **Chapter 6**, we apply ArchOptions in two architecture-centric evolution case studies. We critically discuss and reflect on the strengths and the limitations of its application. We attempt to verify many of the thesis-related claims. We qualitatively evaluate the ArchOptions model and relate its application to the supporting method.

In **Chapter 7**, we summarize the thesis contribution. We highlight possible future research on ArchOptions. We conclude by highlighting some open questions that

could stimulate future research in architectural stability, relating requirements to software architectures, and architectural economics.

In **Appendix A,** we provide background information on COCOMO II (COnstructive COst MOdel) [Boehm 1995], a cost and schedule estimation model.

In **Appendix B**, we provide supporting material related to the case study of using ArchOptions to select stable middleware-induced architecture of Chapter 6.

In **Appendix C**, we provide brief background on Net Present Value (NPV) and Discount Cash Flows (DCF) valuation techniques.

Chapter 2

A Survey of Software Architecture Evaluation Methods

In [Bahsoon and Emmerich, 2003a], we have distinguished between two classes of software architecture evaluation methods:

(i) General-purpose methods that evaluate software architectures for qualities that need to be met by the system, such as performance, security, and modifiability, and

(ii) an emerging class of methods that explicitly evaluate for stability and evolution.

In this chapter, we first review representative examples of (i). The motivation behind this review is to find through existing research stocks insights for evaluating software architectures for stability, which we examine in Chapter 3. Many of the ideas presented relate to the use of software evaluation methods in general. Secondly, we report on research effort related to (ii). We document research motivation and perspectives on architectural stability, as reported in the literature. We discuss why and how to evaluate an architecture for stability. We differentiate between two types of approaches to evaluation; these are retrospective or predictive. We note that methods for evaluating software architectures for stability do not exist, with [Jazayeri, 2002] and our work being the only notable exceptions. Thirdly, we briefly survey research effort on Architectures Description Languages (ADLs) as they

have implications for supporting the evaluation of software architectures. ADLs are languages that provide features for modeling and analyzing software architectures. We show how ADLs can be used in the context of evaluating software architectures in general and the evaluation for architectural stability in particular.

2.1 Architectural Evaluation: A Brief Background

In this section, we lay down the groundwork for evaluating a software architecture: we describe architectural review and evaluation; discuss why and when to evaluate an architecture; who is involved in the evaluation; and list approaches to evaluation.

The architecture of the system is the first design artifact that addresses the *quality goals* of the system such as security, reliability, usability, modifiability, stability, and real-time performance. As the manifestation of early design decisions, the architecture represents those design decisions that are hardest to change [Parnas, 1976] and need to be validated against the quality goals for mitigating risks. *Architecture evaluation* is an activity for developing an assessment of an architecture against the quality goals. It is a form of artifact validation. The evaluation is done with the objective of ensuring that the architecture under question satisfies one or more quality goals. Evaluation also aims to ensure that the architecture is *buildable*. That is, the system can be built using the resources at hand: the staff, the budget, the legacy software (if any), and the time allotted before delivery. From an evolution perspective, architectural evaluation is a *preventive* activity that aims to delay the decay (as referred to by Parnas) and limits the effect of software aging [Parnas, 1996]. Architectural evaluations represent a risk-mitigation effort and are relatively inexpensive [Clements et al., 2002].

Architectural evaluation can be applied at any stage of an architecture lifetime. The classical evaluation of an architecture occurs when the architecture has been specified but before implementation has begun. Users of iterative or incremental life-cycle models can evaluate the architectural decisions made at the end of each iteration or during the most recent architectural cycle. For instance, the Rational

Unified Process (RUP) [Krutchten, 1999] splits the development (evolution) process into four phases. These phases are Inception, Elaboration, Construction, and Transition. The four phases (I, E, C, and T) constitute a development (evolution) cycle and produce a software generation. Under the RUP context, the architectural evaluation can span iteratively and intertwinedly the Inception phase and iterations of the Elaboration phase, and/or can take place at the Life-Cycle Architecture (LCA) milestone. At the LCA milestone, the detailed system objectives and scope are examined; the choice of the architecture is considered; and the major risks are identified.

Early evaluation need not wait until an architecture is fully specified. It can be used at any stage in the architecture creation process to examine those architectural decisions already made and choose among architectural options that are pending. Early evaluations may take the form of *discovery reviews*. A discovery review is a very early mini-review activity. It aims to analyze whatever "proto-architecture" may have been crafted. The output of a discovery review is an "iterated" or a "revised" set of requirements and an initial architectural approach to satisfying them, which is subject in turn to later and iterative evaluation. Note that the architecting process is best conducted iteratively and intertwined through requirements, architecting, and validation

Late evaluation is a form of evaluation for an existing architecture. It takes place when the architecture already exists and the implementation is complete. This occurs when an organization inherits some sort of legacy system and need be integrated with the existing system. The evaluation at this level helps the new owners understand the legacy system, and determine whether the system can be counted on to meet its quality and behavioral requirements.

Clements et al. [2002] provides two rules of thumb on when to hold the evaluation. They suggest i) hold the evaluation when the development team start to make decisions that depend on the architecture; and ii) when the cost of undoing those decisions would outweigh the cost of holding the evaluation.

Generally, architectural evaluation is a human-centered activity. The reviews are typically conducted in the presence of key stakeholders, clients, designers, and the evaluation team. Architecture evaluation may involve "thought experiments", modeling, and walking-through scenarios that exemplify requirements, as well as assessment by experts who look for gaps and weaknesses in the architecture based on their experience. The evaluation may be supported by analytic models, simulation tools, and other architectural analysis means (e.g. parsers, Abstract State Machines, etc). These may be quality-specific, suitable to reason about one quality goal (e.g., performance), or multi-quality goal, suitable for assessing more than one quality goal.

2.2 Research Effort on Architectural Evaluation

In this section, we provide a comprehensive review of software architecture evaluation methods. We trace the evolution of software architecture evaluation methods starting from the early effort by [Parnas and Weiss, 1985] on Active Design Reviews (ADRs) up to the latest existing effort. We describe the evolution of the principles and practices of software architecture evaluation through the following methods: the Software Architecture Analysis Method (SAAM) [Kazman et al., 1994]; the Architecture Trade-off Analysis Method (ATAM) [Kazman et al., 1996]; the Active Attribute-Based Architectural Styles (ABASs) [Klein et al, 1999]; the PASA Software Performance Engineering (SPE) [Smith 1990; Smith and Williams, 2002]; Reviews for Intermediate Designs (ARID) (Clements, 2000); and the Cost Benefit Analysis Method (CBAM) [Kazman et al., 2001].

Effort on architectural evaluation goes back to the seminal work of David Parnas and David Weiss [1985]. Their paper entitled "Active Design Reviews: Principles and Practices" is regarded as the cornerstone to the architectural review/evaluation area. In their paper, Parnas and Weiss expressed one of the fundamental principles behind the architectural evaluation methods: undirected and unstructured design reviews for software design do not work. Their work was motivated by the observations that approaches to design review tend to be spotty, ad hoc, and not repeatable. The common practice was -and still is- to identify a group of reviewer, drop a stack of

read-ahead material on their desk a week or so prior the meeting, haul them in a conference room for a few tedious hours, ask for comments on the material read, and hope for the best [Clements et al, 2000]. The outcome of such practice is predictable and entirely disappointing: failing to uncover any serious problems with the design under consideration and propagating the problem to other phases. Obviously, this is attributed to human nature: participants will not have cracked the read-ahead material until the last minute (if at all), or perhaps they have read to make some intelligent comment. In short, the outcome is an unexercised design artifact.

Parnas and Weiss prescribed a better way. ADRs are contrasted with unstructured reviews in which people are asked to read a document, attend a long meeting, and comment on whatever they wish [Clements and Northrop, 2002]. For validating architectural (and other design) specifications, ADRs are suitable. ADRs are particularly well suited for evaluating the designs of single components before the entire architecture has been solidified [Clements and Northrop, 2001]. ADRs can be used to evaluate an architecture that is still under construction. Reviewers are chosen because of their areas of expertise, not simply because of their availability. Each reviewer is given a questionnaire and/or some exercises to complete. The questionnaires and/or the exercises compel them to use the documentation and think about the architecture. The result is that the artifact being reviewed is actually exercised. For example, an exercise might be, "How would you use the facilities provided by this module to send a message to the user and wait a response?" The reviewer would then be obliged to sketch out the answer in pseudo-code, using facilities described in the design and the documentation.

The Software Engineering Institute (SEI) at CMU has played a notable role in evolving and flourishing the principles and the practices of reviews that address Parnas and Weiss's concerns. They have argued to consider the architecture evaluation as a standard part of the development cycle. With a particular focus on architectural design, the SEI has developed a number of methods. Examples include the Tradeoff Analysis Method (ATAM) [Kazman et al., 1996], the Software Architecture Analysis Method (SAAM) [Kazman et al., 1994), and the Active Review for Intermediate Designs (ARID) [Clements, 2000]. These methods have been applied

for years on dozens of projects of all sizes an in a wide variety of domains. Other SEI methods include the Attribute-Based Architectural Styles (ABAS) [Klein et al., 1999], and The Cost Benefit Analysis Method (CBAM) [Kazman et al., 2002]. The only notable effort outside SEI is the Software Performance Engineering (SPE) [Smith 1990; Smith and Williams, 2002]. We describe the above listed methods in the subsequent sections.

The evaluation using these methods generally identifies what the quality goals of interest are and then highlights the strengths and weaknesses of the architecture to meet the identified goals. These methods either explicitly address a single quality goal or multi-quality goals of interest. Abowd et al. [1996] broadly categorize existing techniques to architectural evaluation as either questioning, measuring techniques, or hybrid. Questioning techniques use scenarios, questionnaires, and checklists, and the like for architectural investigation. Measuring techniques use metrics, simulation, prototypes, or experimentations on running systems. Measuring techniques result in quantitative results. These techniques differ from each other in applicability, but they are all used to elicit discussion about the architecture and increase understanding of the architecture's "fitness" with respect to its requirements. Hybrid techniques may combine both questioning and measuring. The architecture evaluation methods described in this review are generally hybrid; they tend to elicit "discussion" about the architecture using questioning techniques and use some measurements for reasoning.

Conceptually, all the architecture evaluation methods described in this review are active design reviews. They require the participation of experts for their specific stake in the architecture. They pursue a path of directed analysis such as eliciting a specific statements on quality goal(s) that the architecture must meet to be acceptable, and then follow an analytical/measuring path to demonstrate how the architecture satisfies (or does not satisfy) the quality goal(s).

2.2.1 The Architecture Trade-off Analysis Method (ATAM)

The Architecture Trade-off Analysis Method (ATAM) [Kazman et al., 1996] does not only reveal how well an architecture satisfies particular quality goals, but it also provides insight into how these goals interact with each other - how they *trade off* against each other [Clements et al., 2001]. ATAM is a *scenario* based architecture evaluation method. A scenario describes the interaction with the system from the stakeholders' point of view. The ATAM uses three types of scenarios. These are *use case scenarios, growth scenarios,* and *exploratory scenarios.* Use case scenarios describe the typical uses of the completed running system. Growth scenarios represent typical anticipated changes of the system. Exploratory scenarios expose the limits or boundary conditions of the current design; in other words, they tend to expose extreme changes that are expected to "stress" the system.

The input to the ATAM consists of an architecture, the business goals of a system, and the perspectives of the stakeholders involved with the system. The ATAM achieves its evaluation of an architecture by utilizing an understanding of the architectural approach that is used to achieve particular quality goals and the implications of that approach. The quality attributes that compromise system "utility" (e.g. performance, availability, security, modifiability, usability, and so on) are elicited, specified down to the level of scenarios, annotated with stimuli and responses, and prioritized. The scenarios are used for the evaluators to understand the inherent architectural risks, non-risks, sensitivity points to particular quality attributes, and trade-offs among quality attributes.

The ATAM can be used at various stages of development (conceptual, before code, during development, or after deployment). The ATAM is fully described in [Clements et al., 2002].

2.2.2 The Software Architecture Analysis Method (SAAM)

The Software Architecture Analysis Method (SAAM) [Kazman et al., 1994] elicits stakeholder's input to identify explicitly the quality goals that the architecture is intended to satisfy. Unlike the ATAM, which operates around a broad collection of quality attributes, the SAAM concentrates on attributes for modifiability, variability (suitable for product line), and achievement of functionality. The development of SAAM was motivated by the observation that practitioners regularly make claims about their software architectures (e.g. "This system is more robust than its predecessor", "Using CORBA will make the system easy to modify and upgrade") that are untestable [Clements et al, 2001]. SAAM tends to make these claims testable; it replaces claims with quality attributes (like maintainability, modifiability, robustness, flexibility, and so forth) and uses scenarios to operationalize these attributes.

SAAM indicates places where the architecture fails to meet its modifiability requirements and in some cases shows obvious alternative designs that would work better. Like ATAM, SAAM is a scenario-based method. A scenario in SAAM is a brief description of some anticipated or desired use of the system. Scenarios are classified as either *direct* or *indirect* scenarios. Direct scenarios are those scenarios that are directly supported by the architecture, meaning that anticipated use require no modification to the architecture for the scenario to be accommodated. An indirect scenario is one that requires a modification to the architecture to be satisfied; the architect describes how the architecture would need to be changed to accommodate the scenario. When two or more indirect scenarios require changes to a single component of an architecture, they are said to *interact* in that component. Areas of high scenario interaction reveal potentially poor separation of concerns in a component. This indicates that the architecture is not documented to the right level of structural decomposition. The right level of structural decomposition often demands that the decomposed component handles one task at a time, easing both its comprehension and evolution.

The input to SAAM consists of an enumerated set of stakeholder's scenarios that represent known or likely changes that the system will undergo in the future. These scenarios are prioritized and mapped onto the architecture representation. The activity of mapping indicates problem areas in the architecture, where the architecture is overly complex (e.g. if distinct scenarios affect the same component(s)) and areas where changes tend be problematic (e.g. if a scenario causes changes to a large number of components). Bass et al. [1998] and Clements et al [2002] provide a complete description of SAAM.

2.2.3 Active Reviews for Intermediate Designs (ARID)

The Active Reviews for Intermediate Designs (ARID) [Clements, 2000] is a hybrid design review method that combines the philosophy of the Active Design Review (ADRs) with the scenario-based evaluation techniques, such as the ATAM or SAAM. ARID is a method for evaluating subdesigns of partial architectures in their early or conceptual phases. Designs of partial architectures are architectural in nature; they are subdesigns that represent the stepping stones to the full architecture. It aims to validate the suitability of the subdesign being proposed from the point of view of other parts of the architecture. ARID is motivated by the fact that if the architectural subdesigns are inappropriate, then the entire architecture can be undermined. Hence, reviewing a design in its early pre-release stage provides valuable early insights into the design's viability and allows for timely discovery of errors, inconsistencies, or inadequacies.

Note that ADRs are primarily used to evaluate detailed designs of coherent units of software, such as modules or components. It tends to address (i) the sufficiency, fitness, and suitability of the services provided by the design, and (ii) the quality and the completeness of the documentation. ARID can be carried out in the absence of complete documentation. In ARID, the reviewers are the design's stakeholders. The reviewers prepare a set of scenarios. Like ATAM and SAAM, a session is held for scenario brainstorming and prioritization. After scenarios are gathered, a winnowing process occurs. In this process, two or more scenarios that are versions of the same scenario or one that subsumes another are merged. Prioritization is by voting: each

reviewer is allowed to vote up to 30 percent of the number of scenarios. Beginning with the scenarios that have received the most votes, the reviewers craft code or pseudo-code that uses the design to carry each scenario.

2.2.4 Attribute-Based Architectural Styles (ABAS)

Attribute-Based Architectural Styles (ABASs) [Klein and Kazman, 1999] build on *architectural styles* to provide a foundation for reasoning about architectural design. An architectural style is a generic description of an architecture. An architectural style specifies the component types, the topological structure relevant to the specific style, and patterns of data and control interaction among the components. A single architectural style may result in several ABASs, where every ABAS reasons about a specific quality attribute. For example, an architecture with a Client-Server architectural style might have a Security Client-Server ABAS, a Modifiability Client-Server ABAS, a Performance Client-Server ABAS, and so forth. ABAS explicitly associate a reasoning framework (qualitative or quantitative) with an architectural style. The evaluation of an architecture is facilitated by a reasoning framework. The reasoning is based on quality attribute-specific models (e.g. performance, reliability, and maintainability models), which exist in the various quality attribute communities. The reasoning framework may be quantitatively grounded (For example based on rate monotonic analysis, queuing theory, or other metrics) or it may be qualitative in nature (such as checklists, questionnaires, or scenario-based analysis).

For example, Rate Monotonic Analysis of the pipe-and-filter style allows the creation of Performance Concurrent Pipelines ABAS to support the architect in reasoning about worst-case latency quantitatively. Similarly, adding scenario-based reasoning using SAAM, allows the creation of Modifiability Layering ABAS, which supports the designer in reasoning about the effects of changes on the modifiability and maintainability of the system. As far as evaluation is concerned, a style may be "stressed" by stimuli on quality of interest. The objective is to gain insight into the *responses* of the architecture under evaluation to these stimuli using a quality-specific models as a basis of reasoning. The architectural properties are provided as input to

the analysis. This aids the architect in understanding how to achieve a desired response by manipulating the architectural parameters. ABAS facilities evaluating qualities of a generated architectural design and trading among different architectural alternatives.

2.2.5 Software Performance Engineering (SPE) & Performance Assessment of Software Architectures (PASA)

Software Performance Engineering (SPE) is systematic quantitative approach to proactively analyze and manage software performance [Smith, 1990; Smith and Williams, 2002]. The SPE technique can be used to examine an architecture to see whether the designed system will meet it performance constraints. It uses model predictions to evaluate trade-offs in software functions, hardware size, quality of results, and resource requirements. It also includes techniques for collecting data, principles and patterns for performance-oriented design, and anti-patterns for recognizing and correcting common performance problems. PASA, a Method for the Performance Assessment of Software Architectures, is SPA based [Smith, 1990]. Participants in PASA are key developers and project managers. The assessment of the architecture for performance using PASA starts by the identification of critical use cases that are important to the responsiveness or scalability of the system. For each critical use case, the scenarios that are important to performance are identified. Measurable performance objectives are then identified for each key scenario. The architecture is analyzed to determine whether it will support the performance objectives. In the face of a performance discrepancy, the designer has many choices to make: the performance requirements can be relaxed, functionality can be omitted, hardware capability can be increased, or alternatives architectural designs for meeting the performance objectives are recommended. Conceptually, PASA resembles the ATAM, in which the singular quality of interest is performance.

2.2.6 The Cost Benefit Analysis Method (CBAM)

The Cost Benefit Analysis Method (CBAM) [Kazman et al., 2001] is an architecture-centric method for analyzing the costs, benefits, and schedule implications of architectural decisions. The CBAM builds upon the ATAM to model the costs and

benefits of architectural design decisions and to provide means of optimizing such decisions. Conceptually, CBAM continues where the ATAM leaves; it adds a monetary dimension to ATAM as an additional attribute to be traded off. The CBAM consists of the following steps: i) choosing scenarios and architectural strategies (AS); ii) assessing Quality Attribute (QA) benefits; iii) quantifying the Architectural Strategies; iv) costs and schedule implications; v) calculating desirability; and vi) making decisions.

Upon completion of the evaluation using CBAM, CBAM could have guided the stakeholders to determine a set of architectural strategies that address their highest priority scenarios. These chosen strategies furthermore represent the optimal set of architectural investments. They are optimal based upon considerations of: benefit, cost, schedule, within the constraints of the elicited uncertainty of these judgments and the willingness of the stakeholders to withstand the risk implied by uncertainty. To quantify the architectural strategies benefits, stakeholders are asked to rank each AS in terms of its contribution to each quality attribute of -1 to +1. A +1 means that this AS has substantial positive effect on the QA (for example, an AS under consideration might have substantial positive effect on performance) and -1 means the opposite. Each AS can be assigned a computed benefit score from -100 to +100. CBAM doesn't provide a way to determine the cost; it considers that cost determination is a well-established component of software engineering and is outside its scope. The benefits and scores result in the ability to calculate desirability metrics for each architectural strategy. The magnitude of desirability can range from 0 to 100.

2.3 Evaluating Architectural Stability

In this section, we first document research motivation and perspectives on architectural stability, as reported in the literature. We then discuss why and how to evaluate an architecture for stability. Finally, we differentiate between two types of approaches to evaluation; these are retrospective and predictive.

2.3.1 Architectural Stability in Perspective

Ongoing research on the relation between requirements and software architectures has considered the architectural stability problem as an open research challenge and *difficult* to handle [Finkelstein, 2000; Nuseibeh, 2001; van Lamsweerde, 2001; Emmerich 2002]. In particular, Finkelstein [2000] motivated research in architectural stability. Nuseibeh [2001] proposed the "Twin Peaks" model, a partial and simplified version of the spiral model. The cornerstone of this model is that a system's requirements and its architecture are developed concurrently; that is, they are "inevitably intertwined" and their development is interleaved. Nuseibeh advocated the use of various kinds of patterns – requirements, architectures, and designs- to achieve the model objectives. As far as architectural stability is concerned, Nuseibeh had only exposed a tip of the "iceberg" (as referred to by Nuseibeh): development processes that embody characteristics of the Twin Peaks are the first steps towards developing architectures that are stable in the face of inevitable changes in requirements. Nuseibeh noted that many architectural stability related questions are difficult and remain unanswered. Examples include: what software architectures (or architectural styles) are stable in the presence of changing requirements, and how do we select them? What kinds of changes are systems likely to experience in their lifetime, and how do we manage requirements and architectures (and their development processes) in order to manage the impact of these changes? Our work addresses some of these questions.

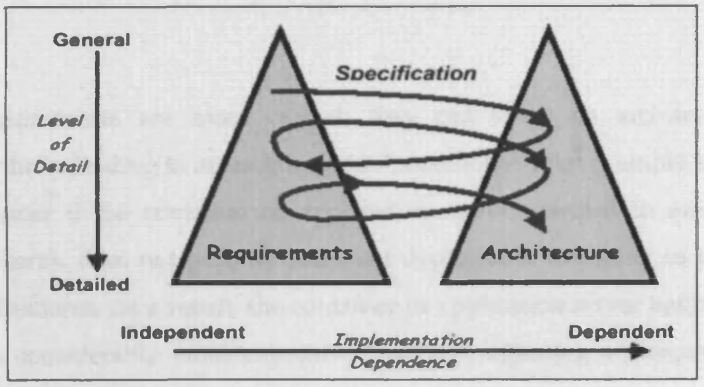


Figure 2.1. Twin Peaks [Nuseibeh, 2001]: a model for the concurrent development of "progressively" more detailed requirements and architectural (design) specifications

Not far from the motivation of bridging the gaps between requirements and software architectures, van Lamsweerde [2000] noted that the goal-oriented approach to requirements engineering may support building and evolving software architectures guaranteed to meet both its functional and non-functional requirements. As far as the architectural stability problem is concerned, van Lamsweerde noted that:

> "Even though streamlined derivation processes may be envisaged for architectural development, things get much more complicated for evolution. For example, the conflict between requirements volatility and architectural stability is a difficult one to handle". [van Lamsweerde, 2000]

Emmerich [2002] has reflected on the architectural stability problem with a particular focus on developing software architectures induced by middleware. Specifically, Emmerich considered the architecture stability problem from the deployment perspective of distributed components technology, in response to changes in non-functional requirements. Emmerich advocates adjusting requirements elicitation and management techniques to elicit not just the current non-functional requirements, but also to assess the way in which they will develop over the lifetime of the architecture. These ranges of requirements may then inform the selection of distributed components technology, and subsequently the selection of application server products. Emmerich considers that addition or changes in functional requirements can be addressed in distributed component-based architectures by adding or upgrading the components in the business logic. However, changes in

non-functional requirements are more critical; they can stress an architecture considerably, potentially leading to an architectural "breakdown". For example, such breakdown may occur if the container or application server, selected to execute distributed components, does not provide sufficient deployment flexibility to meet the changing requirements. As a result, the container or application server has to be changed, which is considerably more expensive than just adjusting a component replication strategy.

In summary, these brief positions have reflected on open challenges and possible strategies in developing software architectures that need to be stable as requirements evolve. They have highlighted the architectural stability problem from a requirements perspective. Focused research attempts, however, have not followed these lines. Hence, the concept is still far from being fully understood and the problem is left unaddressed. Our perspective provides a compromise through linking technical issues to value creation. The approach, which we suggest in this thesis, has the promise to provide insights and a basis for analysis and support for many of the concerns highlighted above. The approach demonstrates that using value-based reasoning, we can analyze for architectural stability and support the development of software systems that need to adapt to inevitable evolving requirements.

2.3.2 Approaches to Evaluating Architectural Stability

Evaluating architectural stability aims to assess the extent to which the system of a given architecture is evolvable, while leaving the architecture and its associated design decisions unchanged as the requirements change. Approaches to evaluating software architectures for stability can be *retrospective* or *predictive* [Jazayeri 2000]. Both approaches start with the assumption that the software architecture's primary goal is to guide the system's evolution. Retrospective evaluation looks at successive releases of the software system to analyze how smoothly the evolution took place. Predictive evaluation provides insights into the evolution of the software system based on examining a set of *likely* changes and the extent to which the architecture can endure these changes.

Retrospective Evaluation

To the author's knowledge, the only visible research effort on architectural stability is the work of Jazayeri [2002]. Jazayeri has looked at the problem from a software evolution perspective. Jazayeri motivated the use of retrospective approaches for evaluating software architectures for stability. Retrospective evaluation looks at successive releases of a software system to analyze how smoothly evolution took place. The analysis relies on comparing properties from one release of the software to the next. The intuition is to see if the system's architectural decisions remained intact throughout the evolution of the system, that is, through successive releases of the software. Jazayeri refers to this "intuitive" phenomenon as architectural stability. Retrospective analysis can be used for empirically evaluating an architecture for stability; calibrating the predictive evaluation results; and predicting trends in the system evolution [Jazayeri, 2002]. In other words, retrospective analysis can also provide a basis for predictive analysis. For example, previous evolution data of the system may be used to anticipate the resources needed for the next release of the system, or to identify the components most likely that require attention, need restructuring or replacements, or to decide if it is time to entirely retire the system. In principle, predictive analysis and retrospective analysis should be combined. However, perfect predictive evaluations would render retrospective analysis unnecessary [Jazayeri, 2002].

Jazayeri's approach uses simple metrics such as software size metrics, coupling metrics, and color visualization (see Figure 2.2.) to summarize the evolution pattern of the software system across its successive releases. The evaluation assumes that the system already exists and has evolved. This approach is therefore not preventive and unsuitable for early evaluation (unless the evolution pattern is used to predict the stability of the next release). The evaluation appears to be expensive and unpractical (in the absence of dedicated tools), for it requires information to be kept for each release of the software. Such data could be available through configuration management repositories. Yet such data is not commonly maintained, analyzed, or exploited. Moreover, as we will see in Chapter 3, the problem of architectural stability is strategic in essence and not purely technical. Jazayeri addresses the problem from a purely technical perspective.

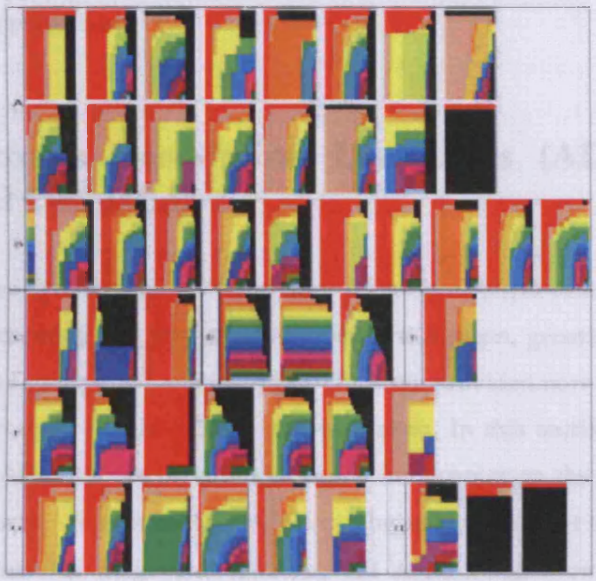


Figure 2.2. Color visualization of module evolution- Jazayeri [2002]

Predictive Evaluation

Retrospective approaches for evaluating architectural stability are unsuitable for early evaluation; the approach assumes that the system already exist and has evolved. The evaluation tends to summarize how smoothly the evolution has taken place. In contrast, predictive approaches can be applied during the early stages of the development life cycle to predict threats of evolution to the stability of the software architecture. Unlike retrospective approaches, predictive approaches are *preventive*; the evaluation aims to understand the impact of the change on the stability of the architecture *if* the *likely* changes need to be accommodated, so corrective design measures can be taken. Therefore, in predictive approaches the effort to evaluation is justified and the evaluation is generally cost effective, when compared to retrospective approaches. Briefly, in ArchOptions (detailed in Chapter 4), we examine a set of *likely* changes that are critical to the evaluation. This begs the question: How can we predict the change? We pursue scenarios as a possible solution to describe these changes. To link the likely future change in requirements to the architecture, we adopt Goal-Oriented Requirements Engineering (GORE) paradigm, where the goals are extracted from scenarios [Anton, 1997]. We then

predict the extent to which the architecture can endure these changes taking a value-based reasoning to prediction.

2.4 Architectures Description Languages (ADLs) and Architectural Evaluation

Although software evaluation methods are typically human-centered, formal notations for representing and analyzing architectural designs, generically referred to as Architectures Description Languages (ADLs), have provided new opportunities for architectural analysis [Garlan, 2000] and validation. In this section, we briefly survey efforts on ADLs as they have implications for supporting the evaluation of software architectures. We explain how ADLs can be used to support the evaluation of software architectures in general and provide some insights on their use to evaluate the architecture for stability in particular.

ADLs are languages that provide features for modeling a software system's conceptual architecture [Medovidovic and Taylor, 1997]. ADLs provide a concrete syntax and a conceptual framework for characterizing architectures [Garlan et al., 1997]. The conceptual framework typically subsumes the ADL's underlying semantic theory (e.g., CSP, Petri nets, finite state machines).

A number of ADLs have been proposed for modeling architectures both within a particular domain and as general-purpose architecture modeling languages [Medovidovic and Taylor, 1997]. Examples are Aesop [Garlan *et al.*, 1995], Darwin [Magee *et al.*, 1995; Magee and Kramer, 1996], MetaH [Vestal, 1996], C2 [Medovidovic *et al.*, 1996], Rapide [Luckham and Vera, 1995], Wright [Allen and Garlan, 1994], UniCon [Shaw *et al.*, 1995], SADL [Moriconi *et al.*, 1995], and ACME [Garlan *et al.*, 1997].

ADLs are often intended to model large, distributed, and concurrent systems. Evaluating the properties of such systems upstream, at the architectural level, can

substantially lessen the costs of any errors. The formality of ADL renders them suitable for the manipulation by tools for architectural analysis. In the context of architectural evaluation, the usefulness of an ADL is directly related to the kinds of analysis a particular ADL tends to support. The type of analyses and evaluation for which an ADL is well suited depends on its underlying semantic model. We refer to Medovidovic and Taylor [1997] to state few examples: Wright is based on CSP; it analyses individual connectors for deadlocks. MetaH and UniCon both support schedulability analysis by specifying non-functional properties, such as criticality and priority. SADL can establish relative correctness of two architectures with respect to a refinement map. Rapide's and C2's event monitoring and filtering tools also facilitate analysis of an architecture. C2 uses critics to establish adherence to style rules and design guidelines.

Another aspect of analysis, that supports architectural evaluation, is enforcement of constraints. Parsers and compilers enforce constraints implicit in types, non-functional attributes, component and connector interfaces, and semantic models. Static and dynamic analyses are used. Static analysis verifies that all possible executions of the architecture description conform to the specification. Static analysis helps the developers to understand the changes that need to be made to satisfy the analysed properties. They span approaches such as reachability analysis [Holzman, 1991; Valmari, 1991; Godefroid and Wolper, 1991], symbolic model checking [Brush et. al, 1990; McMillan, 1993], flow equations, and data-flow analysis [Dwyer and Clarke, 1994]. The applicability of such techniques to architecture descriptions has been demonstrated in [Naumovich *et al.*, 1997] using two static analysis tools. These tools are INCA [Corbett and Avrunin, 1995] and FLAVERS [Masticola and Ryder, 1991; Dwyer and Clarke, 1994]. Rapide [Lukham *et al.*, 1995] provides a support to simulate the executions of the system. The simulation verifies that the traces of those executions conform to high-level specifications of the desired behavior. Allen and Garlan [1994] use the static analysis tool FDR [Formal Systems, 1992] to prove freedom from deadlock as well as compatibility between the component and connectors in an architecture description. The term dynamic architectures denote that application's architecture evolves during runtime. Examples of analyses support for dynamic architectures include the work of [Magee and Kramer, 1996]. Magee and

Kramer's Darwin provides a support to the analysis of distributed message-passing systems.

In the context of evaluating software architectures for stability, no notable research effort has explored the role of ADLs in supporting such evaluation. However, we believe that ADLs have the *potential* to support such evaluation. For instance comparing properties of ADL specifications for different releases of a software can provide insights on how the change(s) or the likely change(s) tends to threat the stability of the architecture. This can be achieved by analyzing the parts of newer versions that represent syntactic and semantic changes. Moreover, the analysis can provide insights into possible architectural breakdown upon accommodating the change. For example, the analysis may show how the change may break the architectural topology (e.g., the architectural style) and/or the architectural structure (e.g., components, connectors, interfaces ect.). We note that ADLs have potential for performing retrospective evaluation for stability. In this context, the evaluation can be performed at a correspondingly high level of abstraction. Henceforth, the evaluation may be relatively less expensive as when compared, for example, to the approach taken by [Jazayeri, 2002], detailed in the previous section.

2.5 Critical Assessment

Architectural evaluation aims at providing confidence that the system of the crafted architecture is buildable, meets both its functional and quality goals (i.e., non-functional requirements), and satisfies the constraints entailed by the environment in which the system works. Table 2.1 depicts a summary of the surveyed general-purpose software architectural evaluation methods. These methods provide frameworks for software architects to evaluate architectural decisions with respect to quality attributes that need to be met by the system. Examples of these quality attributes include performance, security, reliability, and modifiability. Despite the concern with "change" and accommodating changes, some existing architectural evaluation methods focus explicitly on construction and only implicitly, if not at all, on the phenomenon of software "evolution". Further, none of these methods,

addresses stability of an architecture over time. For example, ATAM and SAAM indicate places where the architecture fails to meet its modifiability requirements and in some cases shows obvious alternative designs that would work better. When used for evaluating modifiability, the input to these methods consists of an enumerated set of stakeholders' scenarios that represent known or likely changes that the system will undergo in the future. These scenarios are prioritized and mapped onto the architecture representation. The activity of mapping indicates problem areas in the architecture: areas where the architecture is overly complex (e.g., if distinct scenarios affect the same component(s)) and areas where changes tend be problematic (e.g., if a scenario causes changes to a large number of components). The approaches to evaluation involve "thought experiments", modeling, and walking-through scenarios that exemplify requirements, as well as assessment by experts who look for gaps and weaknesses in addressing modifiability based on their experience. However, these methods do not support their prediction with an analytical basis and rigorous models. When methods, such as SAAM and ATAM are used to analyze qualities that are related to change (such as modifiability), they do not predict and measure the capability of the architecture to withstand the change. This renders their predictive effectiveness myopic. Further, these methods have ignored any economic considerations, with CBAM [Asundi and Kazman, 2001] being the notable exception. The evaluation decisions using these methods tend to be driven by ways that are not connected to, and usually not optimal for value creation. Factors such as flexibility, time to market, cost and risk reduction often have high impact on value creation [Boehm and Sullivan, 2000]. Such ignorance is in stark contrast to the objective of architectural evaluation, where cost reduction, risk mitigation, and long-term value creation are among the major drivers behind conducting evaluation. This brings a need for economics-driven models of predictive power for supporting the evaluation. Such provision is important for "it assists the objective assessment of the lifetime costs and benefits of evolving software, and the identification of legacy situations, where a system or component is indispensable but can no longer be evolved to meet changing needs at economic cost" [Cook et al., 2001].

Table 2.1. A summary of the reviewed general-purpose evaluation methods

Method	Technique	Goals of Interest	Approach to Evaluation	Development/ Evolution
ATAM	Scenario-based	Emphasizes: modifiability, security, and performance	Thought experiments, walk through scenarios, assessment by experts	Development/ Evolution implicit
SAAM	Scenario-based	Modifiability, variability, achievement of functionality	Thought experiments, walk through scenarios, assessment by experts	Development/ Evolution implicit
ARID	Scenario-based	Suitability of functionality	Walk through Scenarios, pseudo-code analysis, assessment by experts	Development
ABAS	Scenario-based; Measuring	Emphasizes: modifiability, security, and performance	Reasoning framework associated with an architectural style to facilitate the evaluation	Development/ Evolution implicit
PASA/SPE	Use-cases/ Scenario-based; Measuring	Performance	Predictive models to evaluate trade-offs in software functions; hardware size; quality of results; and resource requirements	Development
CBAM	Scenario-based; Measuring	See ATAM AND Cost, benefits, Scheduality	Economics-driven; Based on optimizing benefits; costs; and schedule	Development/ Evolution

Despite addressing the costs and benefits of architectural strategies, CBAM does not address stability. Further, CBAM does not tend to capture the long-term and the strategic value of the specified strategy. When CBAM complements ATAM [Kazman et al., 1998] to reason about qualities related to change such as modifiability, CBAM does not supply a rigorous predictive basis for valuing such impact.

We have described research perspectives on architectural stability. We have discussed why and how to evaluate an architecture for stability. We have differentiated between two types of approaches for evaluation; these are retrospective and predictive, as depicted in Table 2.2. We have critically compared the strengths and limitations of these approaches. Retrospective evaluation can summarise how smoothly the evolution took place across releases of the software

system. The evaluation assumes that the system already exists and has evolved making this approach not preventive and unsuitable for early evaluation (unless the evolution pattern is used to predict for the stability of the next release). Evaluation appears to be expensive and unpractical (in the absence of dedicated tools), for it requires information to be kept for each release of the software. Such information could be available through configuration management repositories. Yet, such data is not commonly maintained, analyzed, or exploited. Though using retrospective evaluation it may be feasible to predict future evolvability of an architecture by assessing how easily it evolved in the past, these approaches cannot easily be applied for short and uncertain history [Cook et al., 2001]. In contrast, predictive evaluation provides insights into the evolution of the software system based on examining a set of likely changes and the extent to which the architecture can endure these changes. Unlike retrospective evaluation, predictive evaluations are preventive and can lead to corrective design measures.

Moreover, the problem of architectural stability and its "resilience" over time is strategic in essence and not purely technical. Jazayeri has addressed the problem from a purely technical perspective. Instead, we aim to assist in proactively engineering stable architectures. We believe that the economic interplay between evolving requirements and architectural stability need to be addressed.

Table 2.2. Methods for explicit evaluation for stability and evolution

Method	Technique	Goals of Interest	Approach to Evaluation	Development/ Evolution
Jazayeri's Approach	Quantitative Retrospective	Stability	Retrospective evaluation; design metrics	Evolution Explicit/ Development
ArchOptions	Quantitative Predictive	Stability, Added Value	Predictive evaluation; Economics- Driven; value based reasoning; Real options theory	Evolution Explicit/ Development

Though our current work on ArchOptions does not exploit Architecture Description Languages (ADLs), we have briefly surveyed research effort on ADLs as they have implications on architectural evaluation. The key message is that that role of ADLs is left unexplored in the evaluation of architectural stability. In this context, it is

believed that ADLs can facilitate the evaluation at correspondingly higher level of abstraction than code, as when compared to the approach taken by [Jazayeri, 2002]. Hence, the evaluation may be relatively less expensive.

To address the shortcomings of the surveyed methods, the next Chapter highlights the requirements for evaluating architectural stability from an economics driven software engineering perspective [EDSER 1-7, 1999-2005; Boehm and Sullivan 2000].

Chapter 3

Requirements for Evaluating Architectural Stability

In the previous chapter, we have reviewed research work on architecture evaluation. We have discussed their limitations in addressing architectural evaluation for stability. In this chapter, we state the requirements for evaluating architectural stability when addressed from an economics-driven perspective [EDSER 1-7, 1999-2005; Boehm and Sullivan 2000].

3.1 Requirements for Evaluating Architectural Stability

In a nutshell, if the business goal is that a system should be long-lived, should *evolve* to accommodate future requirements, and should support value creation, it becomes necessary to evaluate the stability of an architecture. The evaluation has to relate technical issues to value creation. The evaluation has to proactively address the economic ramifications of the likely critical changes in requirements and their impact on the architecture. Below, we highlight the requirements that should be addressed when evaluating an architecture for stability.

Assess Evolution

Despite the concern with "change" and accommodating changes, existing architectural evaluation methods focus explicitly on construction and only implicitly, if not at all, on the phenomenon of software "evolution". A Software architecture represents those design decisions that are hardest to change [Parnas, 1996]. From an

evolution perspective, architectural evaluation is a *preventive* activity that aims to delay the decay and limit the effect of software aging [Parnas, 1996]. Easing evolution is the underlying, if implicit, motivation for many of the recent software development practices, which place considerable emphasis on the architecture of the software system as the key artifact involved. For example, product-line architectures aim at the systematic controlling of software evolution [Jazayeri, 2002]. Product-line architectures anticipate the major evolutionary milestones in the development of the product, capture the properties that remain constant through evolution and document variability points from which different family members may be created. The approach gives a structure to the products' evolution and possibly rules out some unplanned evolutions, if the architecture is respected [Jazayeri, 2002]. Though the software architecture, as a key designed artifact, is considered to be "the promising solution for easing software maintenance and evolution" [Jazayeri, 2002], rapid technological advances and industrial evidence are now showing that the architecture is creating its own maintenance, evolution, and economics problems. For example, assume that a distributed e-shopping system architecture which relies on a fixed network needs to evolve to support new services, such as the provision of mobile e-shopping. Moving to mobility, the transition may not be straightforward: the original distributed system's architecture may not be respected, for mobility poses its own non-functional requirements for dynamicity that are not prevalent in traditional distributed setting [Capra, 2003]. Examples of these requirements include the need to react to frequent changes in the environment, such as change in location, resource availability, variability of network bandwidth, the support of different communication protocols, loss of connectivity when the host need to be moved, and so forth. These requirements may not be satisfied by the current fixed architecture, the built-in architectural caching mechanisms, and/or the underlying middleware. Replacement of the current architecture and/or its underlying middleware may be required.

Therefore, what constrains the success of evolving the system with a given architecture is the ability of the architecture to support the likely change in requirements. In evaluating architectural stability, the architectural evaluation may not only need to assess how the current requirements could be realized by the

architecture, but also the ranges in which these requirements may change and evolve during the life time of the software system.

"Continual" Investment Management in an Architecture

According to Bennet and Rajilich [2000], software evolution takes place only when the initial development was successful. The goal is to adapt the system to the changing requirements. The inevitability of evolution is documented in [Lehman, 1985]: "the software is being evolved because it is successful in the marketplace, revenue streams are buoyant, user demand is strong, and the organization is supportive. Return on investment is excellent". Hence, evolution is primary driven by business needs. Conversely, software evolution needs to seek and create value relative to the resources invested [Bahsoon and Emmerich, 2004a]. As such, the costs of evolving software should not outweigh the returns from the process to achieve a net benefit. Under the assumption that the primary role of the software architecture is to guide evolution, the success of software evolution is hence dependent on the architecture [Bahsoon and Emmerich, 2004a; Bahsoon and Emmerich, 2004b]. An architecture needs to be *flexible enough* to accommodate the change(s) without breaking the architecture itself, the supporting infrastructure, and/or the topology. Breaking the architecture is costly. On the other hand, having an "overly flexible" architecture implies upfront costs, which could not be utilized to achieve a net benefit [Bahsoon and Emmerich, 2004a; Bahsoon and Emmerich, 2004b]. For example, if a likely change does not occur, then the value of the system decreases because the flexibility will not pay off. When a likely change is significant enough, the architecture needs to be considered to incorporate enough flexibility with the promise that such flexibility could lead to the right to claim future cost savings. Accounting for evolution brings a need for continuous "management" and optimization for the net benefit of the flexibility provided by the architecture. This needs to be considered upon evaluating an architecture for stability. As the success (failure) of evolution is very much linked to the architecture, the long-term costs and likely savings are revealing measures to the "resilience" of the architecture to the change. The ability of a system with a given architecture to maintain/add value as the software system evolves is hence indicative of its stability.

In short, "designing for change" is approached by designing flexible and customizable architectures. One of the major criteria that the architectural evaluation for stability should consider is optimizing the net-benefit of the embedded or the adopted flexibility of the architecture relative to the likely changes. A particular question of interest is: how much to do we need to invest in designing for change and how valuable are the associated design decisions? Investing in flexibility incurs upfront costs and may render future and long-term benefits, such as supporting software reuse and instantiating from the core architecture new market products. Hence, the tradeoff between the upfront investment and the long-term future benefits should be assessed.

Strategic Considerations

In software engineering, the term strategy refers to techniques that treat uncertainty, incomplete knowledge, risk, competition, and related issues systematically, consciously, and in a sound manner with the aim to maximize the expected value of a given product or project [Sullivan et al., 1997]. Strategic considerations are related to or concerned with strategy. The term implies that the focus is on improving and sustaining the "performance" of the software system over time in meeting both its technical and business goals, aligning the system and its evolution with the organization's performance objectives, and seeking new strategic opportunities. We consider the architecture as the appropriate level of abstraction at which to think of strategic software decisions and guide the evolution of the software system. Further, the problem of architectural stability and the architecture "resilience" to evolution is strategic in essence and not purely technical [Bahsoon and Emmerich, 2003a]. A stable architecture is a significant strategic asset during the operation and the evolution of the software system. Stability is an architectural quality with strategic importance and with long-term strategic and operational benefits. Stability is said to be of strategic importance as it reflects on the architecture's "performance" over time, the architecture "dynamism" with respect to likely changes in requirements over the projected life of the software system, and its "resilience" to change(s). Architectural stability may result in benefits of strategic importance, such as the opportunity to instantiate from the architecture new market products; the flexibility to respond to competitive forces and changing market conditions; and the ability to accommodate new services. It may also render long-term operational benefits, such as reduced

maintenance cost. A characteristic of these benefits, whether strategic or operational, is that their payoffs are uncertain and may not be immediate.

Our consideration of stability as a strategic architectural quality reveals a new segmentation of architectural qualities, which appears to be absent from the software architecture literature. For example, Bass, Clements, and Kazman [1997] segment architectural qualities into two: these are dynamic (i.e., qualities observed via execution, e.g. performance) or static (i.e., qualities not observed via execution, e.g. modifiability). Both segments correspond to qualities, which need to be "built" into the software to fulfill its requirements. Even when qualities such as modifiability are considered under Bass and Clements' segmentation, they are treated from a "build" perspective as opposed to an investment. However, stability poses challenges, which make it difficult to be considered under Bass and Clements' segmentation of architectural qualities. Intuitively, the stability term refers to the "resilience" of an entity over a time period in the face of changes. The term implies a time dimension; it necessitates observing the effect of the change on the "global" properties of the subject architecture relative to its predecessor(s). The "global" properties may not necessarily be structural or behavioral; they may "crosscut" the business goals and other factors that constrain the architectural decisions.

In this context, evaluating an architecture for stability must address the following strategic dimensions: (i) the time-line in which likely changes may need to be realized; (ii) the long-term cost of accommodating the change; and (iii) the long-term value implications of the architectural potential in accommodating the change.

Addressing Uncertainty

Uncertainty is defined as an event that can happen, but the probability of its occurrence is unknown [Ross et al., 1996]. We identify three major types of uncertainty, which need to be addressed upon evaluating an architecture for stability. First, the uncertainty associated with the change, its complexity, and its likelihood. The change could be considered as a major source of uncertainty that may place the investment in a particular architecture at risk. For example, the uncertainty

might be because of changes in stakeholders preferences and expectations, features of a system that are likely to change in the future and across a product line, changes in the environment in which the system works, macroeconomic influences, organizational changes, new market demands such as standardization, internationalization, product segmentation, economics constraints and so forth. These changes may not necessarily be perceived during the development of the software system. Second, when the change in requirements is likely, the value that the analysts ascribe to the architecture in supporting the change, perhaps resulting in new products, is often uncertain. Uncertainty of this value implies variation in the probable future values of the "architectural potential" relative to the change. Third, even if the changes in requirements are perceived during the development, the *You Aren't Going to Need It* principle (YAGNI) [http://xp.c2.com/], for example, may entail delaying the implementation of some of these requirements until uncertainty about their value is resolved. When applicable, this means that the evaluation shall also address the value of delaying an investment decision in the change and relative to the uncertainty of the requirement's value itself. Fourth, the uncertainty which is partially driven by the immaturity of the discipline and the state-of-practice in eliciting requirements, anticipating their changes, the way the change relate to the architecture, and the unique nature of the architecture as a capital asset. Unfortunately, there are no silver bullets, that can address these challenges. Yet, we believe that architectural evaluation for stability should try to control these uncertainties as much as possible in order to mitigate risks.

Architectural Integrity

An architecture with limited flexibility may realize the change through "cosmetic" solutions of an ad-hoc or propriety nature, such as modifying part of the architecture; implementing additional interfaces; extending the primitives of the underlying middleware; and so forth. These solutions could be costly, problematic, and unacceptable. Yet these solutions may turnout to be more cost-effective in the long-run and relative to other alternatives. Even, if we accept the fact that modifying the architecture or the infrastructure is the only solution towards accommodating the change, analyzing the impact of the change and its economics becomes necessary to see how much we are expending to "re-maintain" or "re-achieve" architectural

stability relative to the likely change(s) [Bahsoon et al., 2005]. Though it might be appealing to intuition that the "intactness" of the structure is the definitive criteria for selecting a "more" stable architectures, the practice reveals a different trend: it boils down to the potential added value upon exercising the change. Hence, under some circumstances breaking the architecture could be acceptable [Bahsoon et al., 2005]. Therefore, upon evaluating an architecture for stability, a tradeoff between the architectural "intactness" and the cost-effectiveness of amending the architecture to accommodate the change must be addressed.

3.2 Summary

We have highlighted the requirements for evaluating architectural stability from an economics-driven software engineering perspective. These requirements entail finding an approach for assessing evolvability. The approach shall aim at assessing the economic ramifications of the likely critical changes in requirements and their impact on the architecture of the software system, the "profitability" of evolution, and consequently the success of evolution. The approach shall provide the basis for analyzing many of the economic tradeoffs involved in designing and reengineering for the change. Examples include (i) the economic tradeoff between the upfront cost of enabling the change on the architecture of the software system and the resulting long-term future benefits, and (ii) the economic tradeoff between the architectural integrity and the cost-effectiveness of amending the architecture to accommodate the change.

Chapter 4

ArchOptions: A Model for Evaluating Architectural Stability with Real Options Theory

In the previous chapter, we have highlighted the requirements for evaluating software architectures for stability. In this chapter, we pursue an economics-driven approach to address these requirements. We describe a novel model that exploits options theory to evaluate architectural stability. The model is referred to as ArchOptions [Bahsoon et al., 2005; Bahsoon and Emmerich, 2004a; Bahsoon and Emmerich, 2004b; Bahsoon 2003; Bahsoon and Emmerich, 2003b]. The model provides "insights" into the evolution of the software system based on valuing the *extent to which* an architecture is flexible enough to endure some *likely* critical changes in requirements. The model builds on Black and Scholes[1973] financial options theory (Nobel Prize winning) for valuing this flexibility. The valuation provides a basis for analyzing the stability and investment decisions for many architecture-centric approaches to evolution.

We first provide background on real option theory that is necessary to understand our approach. We then describe the options-based approach to the systematic evaluation of architectural stability, leading to the ArchOptions model. We show how we have derived the model, the analogy and the assumptions that the model makes, the model formulation and its sensitivity, and we report on its possible interpretations and usage scenarios. We finally provide an overview of closely related work on the use of real options in software design and engineering.

4.1 Real Options: A Brief Background

Definition

Central to the real options approach is the concept of an option. An option is an asset that provides its owner the right without a symmetric obligation to make an investment decision under given terms for a period of time into the future ending with an expiration date [Schwartz and Trigeorgis 2000]. If conditions favorable to investing arise, the owner can exercise the option by investing the strike price defined by an option. A *call option* gives the right to acquire an asset of uncertain future value for the strike price. A *put option* provides the right to sell an asset at that price. A European option can only be exercised on the expiration date of the option. A *real option* is an option on non-financial (real) asset, such as a parcel of land or a new product design.

What Problems Do Real Options Address?

Real options theory addresses the problem that investment valuation based on discounted cash flow (DCF) and net present value (NPV) tend to overlook the value of decision flexibility. Critics recognize that DCF and NPV often undervalue investment opportunities, leading to myopic decisions, underinvestment, and eventual loss of competitive position. The problem originates in the inability of these techniques to properly value important strategic considerations and to capture the value of future operating flexibility associated with many projects. Myers [1987] acknowledged that these techniques have inherent limitations when it comes to valuing investments with significant operating or strategic options, for they overlook the sequence of interdependence among investments over time. Myers [1987] suggested that options pricing holds the best promise to value such investments.

The options pricing approach has two major advantages. First, it relieves the decision-maker from having to forecast cash flows and predict the probabilities of future states. Second, it provides valuations that are not based on subjective, questionable parameter values, but rather on data from the market or market-calibrated data. In a nutshell, the decision-maker provides the current value of the asset under consideration and the variance in the value over time. That is enough to

determine the "cone of uncertainty" in the future value of the asset, rooted as its current value and extending over time as a function of volatility. The variance is obtained by identifying assets in the market that are subject to the same risks as the one in question. Valuing flexibility using options considers that the risk (variance) is implicit in the asset being considered be "in the span of the market".

Origin

The real options field opened in 1977 when the economist Myers noted that "part of the value of the firm is accounted for by the present value of the options to make further investments on possibly favorable terms" [Myers, 1977]. Myers saw that, all else equal, a firm that is in a position to exploit lucrative opportunities, for example, through an upfront strategic investment, is worth more than a firm that is not. Myers saw that such opportunities take the forms of real (as opposed to financial) options. Real options theory is an emerging field and based on financial options theory. Financial options have been studied since 1900; however, the seminal modern results, which provided long-sought closed-form mathematical formulations for valuing financial options, are due to Black and Scholes [1973], and Merton [1973]. Black and Scholes received the 1997 Nobel Prize in economics for their work on the topic. Many other results, which are now elements of basic finance, have been produced since (e.g., [Brealey and Myers, 1996; Cox and Rubinstein, 1984 and 1979; and McDonald and Siegel, 1986]). For the past 25 years, researchers have been building the theory of real options (e.g., [Brealey and Myers, 1996; Dixit and Pindyck, 1994; Trigeorgis, 1995]).

Real Options Valuation

Options are valued using a variety of techniques. These techniques make different assumptions and require different tools to capture uncertainty. Uncertainty is often captured by a certain stochastic model that represents the movement of the underlying asset value over time. The options valuation determines the value of a project or investment opportunity from the values of other market-traded assets. The quantitative origins of real options derive from the seminal work of Black and Scholes [1973] in pricing financial options. Subsequently, Cox, Ross and Rubinstein [1979] developed a binomial approach that enables a more simplified valuation of

options in discrete time. The mechanics for calculating the value of an option reduce to folding back a decision tree, as done for either a dynamic DCF analysis or decision analysis [Schwartz and Trigeorgis, 2000]. The difference among these techniques revolves around how one chooses relevant values and represents uncertainty. Option pricing focuses on market value and uses the standard deviation of the rate of return on an underlying or (twin asset). The underlying asset is an asset with the same risks as the project (or asset) the firm would own if the options were exercised, that is, if the investment were made and the project completed.

Types and Applications

Real options analysis has been extensively applied to various sectors such as natural resources (exploration and development), pharmaceutical (drug development), real estate (leasing decisions), manufacturing systems (convertible plants), aerospace (aircraft development and acquisition), and information technology (R&D, technology valuation). For examples, see [Schwartz and Trigeorgis 2000] and [Amram and Kulatalika, 1999]. The application of real options in software engineering is detailed in Section 4.4 of this chapter. In traditional applications, real options analysis recognizes that the value of the capital investment lies not only in the amount of direct revenues that the investment is expected to generate, but also in the future opportunities flexibility creates. These include abandonment or exit, delay, exploration, learning, and growth options. The economic literature analyses many types of real options. These real options could either occur naturally in a particular project/real asset (e.g., the option to defer, to contract, to shutdown, or to abandon) or could be planned and built in at some upfront extra cost (e.g., the option to expand capacity, to build growth options, to default when investment is staged sequentially, or to switch between alternative inputs or outputs).

4.2 Architectural Stability: An Options Perspective

In the previous chapter, we have highlighted the requirements for evaluating architectural stability. These requirements necessitate finding an approach, which assesses evolvability and traces technical issues to value creation. The approach shall continually "manage" the investment in evolvable architectures and provide a basis for analyzing the economics of an architectural flexibility in relation to change; the

J2EE application server. A notable difference between these two architectures will be that increasing load demands might be easily accommodated in the J2EE architecture because J2EE application server provide primitives for replication of Enterprise Java Beans that can be used, while the CORBA-based architecture may not easily scale. The choice is not straightforward as the J2EE-based infrastructures usually incur significant upfront license costs. Thus, when selecting an architecture, the question arises whether an organization wants to invest into an J2EE application server and its implementation within an organization, or whether it would be better off implementing a CORBA solution. Answering this question without taking into account the *flexibility* that the J2EE solution provides and how *valuable* this flexibility will be in the future relative to the likely changes in non-functional requirements might lead to making the wrong choice.

In general terms, means for achieving flexibility are typical architectural mechanisms or strategies built-in or adapted into the architecture with the objective of facilitating evolution and future growth, in response to changes in functional (e.g., changes in functionality) or non-functional requirements (e.g., changes in scalability demands). Unfortunately, built-in or adapted flexibility comes with a price. Questions of interest, however, are how worthwhile is it "buying" flexibility to facilitate future changes and support the development (evolution) of potentially stable architectures? How can we select an architecture which maximizes the yield of such flexibility relative to the likely changes in requirements? When does investing in flexibility result in potential stability? We aim to provide an answer to these questions using "options thinking".

Why a Real Options Perspective?

Real options theory is well suited to address many Software Engineering problems from a value-based engineering perspective [Boehm and Sullivan, 2000; EDSER 1-7, 1999-2005]. To understand the stability of software architectures using an economic approach, we need a valuation technique that is suitable for strategic and long-term valuation, accounts for flexibility, and makes the value of the options created by

flexibility tangible, as a way to make the value of stability tangible. Real options satisfy these requirements.

First, real options theory provides an analysis paradigm that emphasizes the value-generating power of flexibility under uncertainty [Erdogmus et al., 2002]. In traditional applications, real options analysis recognizes that the value of the capital investment lies not only in the amount of direct revenues that the investment is expected to generate, but also in the future opportunities flexibility creates. The flexibility may take the form of abandonment or exit, delay, exploration, learning, and growth options. In an evolutionary context, the change is uncertain as the demand on the future changes in requirements is uncertain. Thus, the value-generation of the architectural flexibility in accommodating the change is a powerful heuristic for analyzing investment decisions and its implications on the stability of an architecture. We view stability as a strategic architectural quality that adds to the architecture values in the form of *growth options*. A growth option is a real option to expand with strategic importance [Myers 1987]. Growth options are common in all infrastructure-based or strategic industries with multiple-product generations or applications [Schwartz and Trigeorgis 2000]. As many early investments can be prerequisites or links in chain of interrelated projects [Myers 1987], growth options set the path for the future opportunities. Obviously, investments in software architectures are infrastructure-type of investments. The architecture may provide both the system and the enterprise the potentials for growth. In the architectural context, growth opportunities are linked to the flexibility of the architecture to respond to future changes. Note that flexibility has a value under uncertainty [Ross et al., 1996]. Since the future changes are generally unanticipated, the value of the growth options lies in the enhanced *flexibility* of the architecture to cope with uncertainty; otherwise, the change may be too expensive to pursue and/or opportunities may be lost.

Second, the search for a potentially stable architecture requires finding an architecture that maximizes the yield in the added value, relative to some future changes in requirements. As we are assuming that the added value is attributed to flexibility, the problem becomes maximizing the yield in the embedded or adapted

flexibility in a software architecture relative to these changes. A Real options approach is a value-maximizing paradigm and suited to address this problem. Back to our motivating example, the choice of inducing the architecture with either CORBA or J2EE is a value-maximization problem. What we need to maximize is the added value as a result of choosing either CORBA or J2EE: once a particular middleware is chosen, it will be extremely expensive to revert the choice and adopt a different middleware. As the middleware is responsible for realizing much of the non-functionality, the choice is influenced by the non-functional requirements. Unfortunately, these requirements tend to be unstable and evolve over time. Hence, the choice has to maximize the value added upon accommodating the change in non-functionality, such as the changes in the likely future load. Interested reader may refer to Chapter 6 for an example.

Third, classical financial valuation techniques, such as Discounted Cash Flow (DCF) analysis and Net Present Value (NPV) (see Appendix C for a brief background), fall short in dealing with flexibility and uncertainty [Schwartz and Trigeorgis 2000]. The main problem with these techniques is that they are best valid when valuing an ongoing business or an immediate investment. However, in the case of valuing the stability of software architectures in the face of evolutionary changes, the nature of the investment is long-term and strategic. For example, assume that an investment in an architecture appears to be unattractive, as it would have a negative NPV in the first instance: unless the enterprise makes the initial investment, subsequent generations or other applications will not even be feasible. The value of the investment, thus, may derive not only from the direct measurable cash flows of the investment, but also from the ability of an architecture to unlock future growth opportunities (e.g. case of reuse, exploring new markets, expanding the range of services while leaving the architecture intact).

4.3 The ArchOptions Approach: Valuing Architectural Stability with a Real Options Analogy

In subsequent sections, we describe a real options-based approach for evaluating architectural stability using an analogy with Black and Scholes [1973] options theory.

We describe the approach. We present the analogy. We formulate and interpret the ArchOptions model. We report on its sensitivity and on its possible uses. We discuss valuation issues and assumptions under ArchOptions.

The Approach

We assume that the software architecture's goal is to guide the system's evolution. We view evolving software as a value-seeking and value-maximizing activity: software evolution is a process in which software is undergoing a change (an incremental) and seeking value [Bahsoon and Emmercih, 2004b]. We attribute the value to the flexibility of the architecture in responding to the change(s). In this perspective, we rely on intuition in relating flexibility to stability: flexibility is a strategic resource that is built-in or adapted into the architecture with the aim of facilitating future growth and evolution with the objective of creating value. For example, upon reengineering an architecture to facilitate future changes, the reengineering activity aims at adapting further flexibility into the architecture of the software system. The reengineering exercise may lead to a "more" flexible structure with different value potentials, as depicted in Figure 4.1. The investment in reengineering may create future value. This is because reengineering adapts flexibility into the architecture making it more adaptable than the original version. The realized value may span several dimensions including savings in the future maintenance effort. The value may be realized only if some future changes need to be accommodated on the system of the given architecture. The more valuable the adapted flexibility is in responding to future changes, the more successful the software evolution is likely to be. Consequently, the better the potentials are for maintaining architectural stability. However, in case of an existing architecture with built-in flexibility, the embedded flexibility could be unutilized but may translate into value upon exercising the flexibility as the inevitable change(s) in requirements materializes. Hence, stability is a result of the success (failure) of the flexibility resource in responding to the change(s).

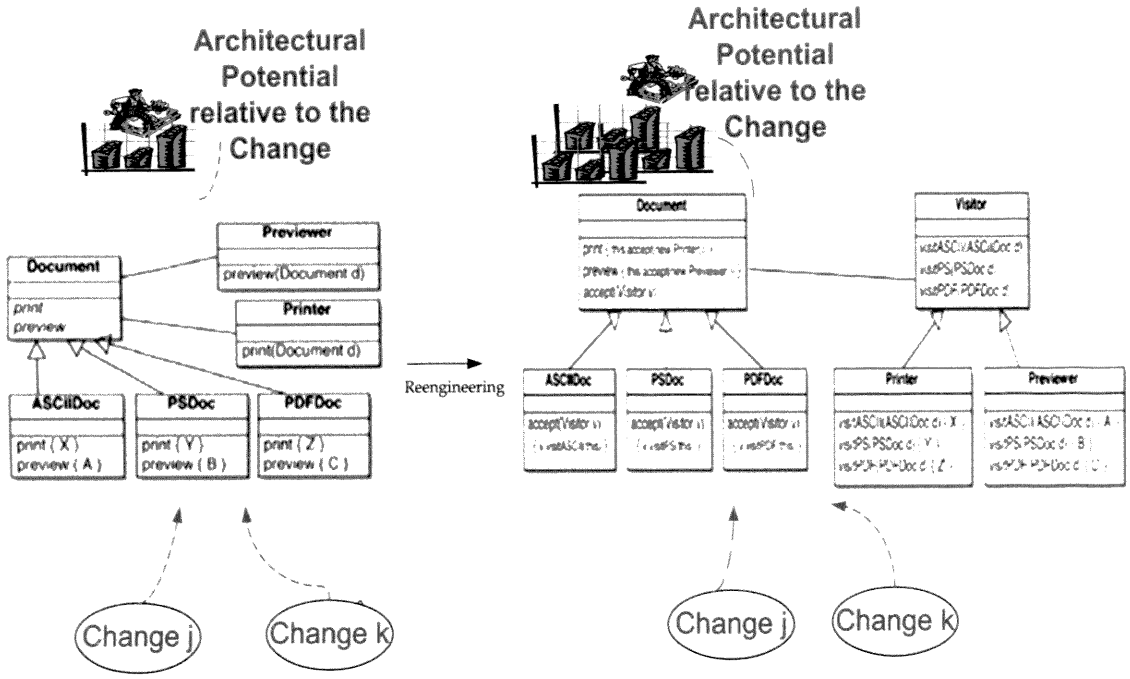


Figure 4.1. Reengineering leading to a "more" flexible structure with different architectural and value potentials upon accommodating some likely change in requirements.

We claim that stable software architectures add to the software system and to the enterprise owing the architecture a value. The added value is attributed to flexibility and the options that flexibility creates over the evolutionary periods of the software system. The added value under the stability context is strategic in essence and may not be immediate. It takes the form of (i) accumulated savings through enduring the change without "breaking" the architecture; (ii) shortened time-to-market through rapid adaptation of new features or requirements and henceforth preserving the competitive position of the enterprise; (iii) savings and opportunities due to reuse; (iv) enhancing the opportunities for strategic "growth" (e.g. regarding an architecture as an asset and instantiating the asset to support new market products); and (v) giving the enterprise a competitive advantage by activating the stable architecture like any other capitalized asset.

In this context, the flexibility of an architecture to endure changes in stakeholders' requirements and the environment has a value that can *assist* in predicting the

stability of software architectures. More specifically, flexibility adds to the architecture values in the form of *real options* that give the right but not a symmetric obligation to evolve the software system and enhance the opportunities for strategic growth by making future follow-on investments (e.g., case of reuse, exploring new markets, expanding the range of services, etc.). In software systems, the change in requirements is a major source of uncertainty that confronts the architecture during its lifetime. As flexibility has a value under uncertainty, the value of these options lies in the enhanced flexibility to cope with uncertainty. The importance of the idea cannot be overemphasized: it gives the architect an ability to reason about a crucial but previously intangible source of value and to employ it in the evaluation of architectural stability.

We contribute to an approach for evaluating the stability of software architectures with real options theory. As we have mentioned in an earlier chapter, approaches to evaluating software architectures for stability can be *retrospective* or *predictive* [Jazayeri, 2002]. We contribute to a predictive approach, where we use value-based reasoning to prediction (real options theory). We examine critical *likely* changes in requirements and *value* the extent to which the architecture is flexible in enduring these changes. These changes could be of functional or non-functional nature.

We derive a predictive model from [Black and Scholes 1973] financial options theory. The model is referred to as ArchOptions. ArchOptions builds on a simple and intuitive analogy with Black and Scholes [1973]. ArchOptions looks at investment in a particular architecture as upfront investment plus future investments in likely future change(s) in requirements. However, these changes are uncertain, as the demand for the change(s) is uncertain. Uncertainty attributed to the change and its likelihood is one of the major reasons, which justify the use of real options theory. For a likely change in requirements, the model constructs a call option to value the flexibility of the architecture to accommodate the change, as a way to make the value of stability tangible. Recall, a call option gives the right to acquire an asset of uncertain future value for the exercise price. Accommodating the change, thus, is analogous to buying an "architectural potential" (i.e., an option on an asset) with uncertain future value paying an exercise price. The exercise price corresponds to the

cost of accommodating the change on the system of the given architecture. The value of the call option, whether in-the-money or out-of-the-money, is a measure of the architecture flexibility in accommodating change. This value is an indicative measure of the "architectural potential" in unlocking future growth opportunities (e.g., case of reuse, new market products), enhancing the upside potentials of the architecture, generating value (e.g., savings in maintenance), or incurring losses (e.g., case of a disruptive changes), as a consequence of accommodating the change. The value of the call is a powerful heuristic, which can provide a basis for analyzing many architecture-centric evolution problems, which place considerable emphasis on the flexibility of the architecture as a way for easing software evolution. For example, the value can provide insights into the economics of flexibility, the inflexibility, and the over-flexibility of the architecture relative to the change. The value of the calls can have extensive uses as highlighted in Section 4.3.

As the values of the calls are correlated with the extent to which an architecture is flexible, whether this flexibility is embedded or adapted, the search for a potentially stable architecture requires finding an architecture or an associated artifact, which maximizes the yield in the calls relative to some critical changes.

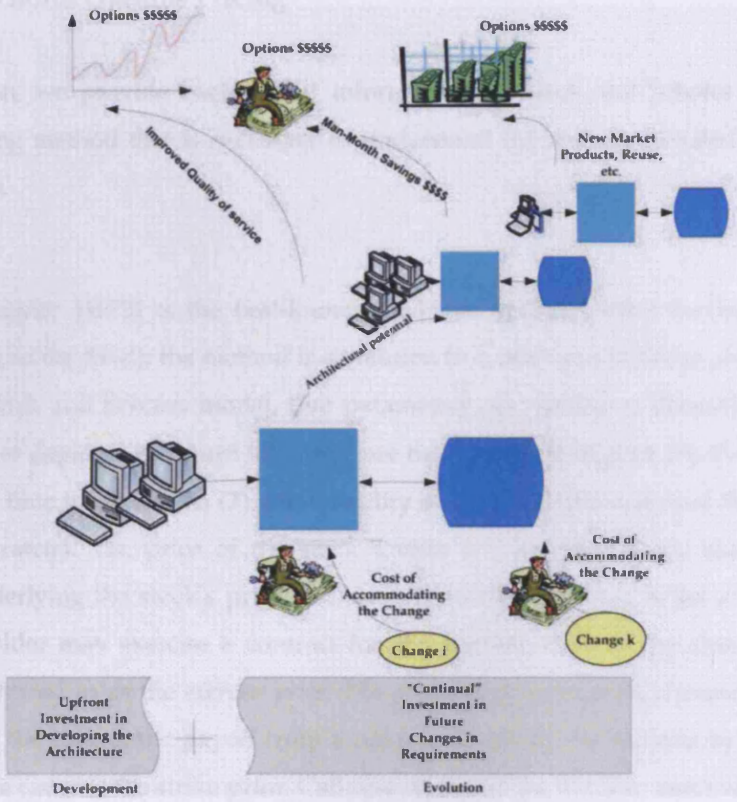


Figure 4.2. The model looks at an investment in an architecture as an upfront investment plus increments of future investments in some likely changes in requirements.

In brief, the approach considers the architecture as the appropriate level of abstraction at which to think about strategic investment decisions, guide the evolution of the software system, and analyze the evolution value, costs, and investment opportunities. The approach builds on a sound theory in financial engineering to provide "insights" into architectural stability, investment decisions related to the evolution of software architectures, and a basis for analysis for many architecture-centric evolution problems, with desired stability requirements.

Black and Scholes Options Pricing

In this section, we provide background information on Black and Scholes [1973] options pricing method that is necessary to understand the analogy detailed in the next Sections.

Black and Scholes [1973] is the best-known financial option pricing method (the seminal work in the field); the method is a solution to a stochastic calculus problem. Under the Black and Scholes model, five parameters are needed to determine the option price, as depicted in Figure 4.3. These are the current stock price (S), the strike price (X), the time to expiration (T), the volatility of the stock price (σ), and the free-risk interest rate(r). The price of the stock option is a function of the stochastic variables underlying the stock's price and time. The strike price (X) is the price for which the holder may exercise a contract for the purchase/sale of the underlying stock; also referred to as the *exercise price*. The current stock price (S) if exercised at some time in the future, the payoff from a call option will be the amount by which the stock price exceeds the strike price. Call options, therefore, become more valuable as the stock price increase and less valuable as the strike price increases. The volatility of the stock price (σ) is a statistical measure of the stock price fluctuation over a specific period of time; it is a measure of how uncertain we are about the future of the stock price movements. The value of a call option on an asset depends on the value of the asset itself and the cost of exercising the option.

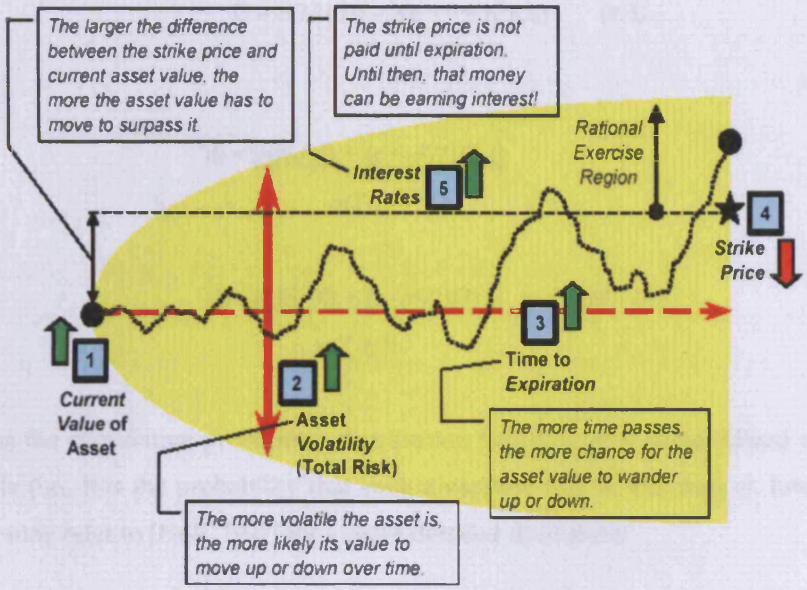


Figure 4.3. Five Parameters determining the value of call options [Erdogmus et al., 2002]

The expected value of a European call option is given by E [max (S_t- X, 0)], where E denotes the expected value of a European call option and S_t denotes the stock price at time t. The European call option price, C, is the value discounted at the risk-free rate of interest. It calculates to equation *(4.1)*.

$$C = e^{-r(T-t)} E\,[\max(S_t - X, 0)] \qquad (4.1)$$

In a risk-neutral world, ln S_t has the following probability distribution given by *(4.2)*,

$$\ln S_t \sim \phi\,[\ln S + (r-\sigma^2/2)(T-t),\ \sigma(T-t)^{1/2}] \qquad (4.2)$$

where ϕ [m, s] denotes a normal distribution with mean m, and standard deviation S. Evaluating the right-hand side of *(4.1)*- in application of integral calculus- results in Black and Scholes valuation of a European call option.

$$C = S\,N(d_1) - Xe^{-r(T-t)}\,N(d_2) \qquad (4.3)$$

where,

$$d_1 = \frac{\ln(S/X) + (r + \sigma^2/2)(T-t)}{\sigma(T-t)^{1/2}}$$

$$d_2 = \frac{\ln(S/X) + (r - \sigma^2/2)(T-t)}{\sigma(T-t)^{1/2}} = d_1 - \sigma(T-t)^{1/2}$$

$N(x)$ is the cumulative probability distribution function for a standardized normal variable (i.e., it is the probability that such a variable will be less than x). Interested reader may refer to [Hull, 1997] for a more detailed derivation.

The Analogy

A major insight behind real options theory is that flexibility in real asset is analogous to financial options: investing in flexibility is said to be analogous to creating options on an asset and exercising such flexibility is seen as exercising options for buying. Having set the flexibility of the architecture in responding to likely changes in requirements as an option problem, the challenge becomes valuing such flexibility. We build on a simple and intuitive analogy with Black and Scholes [1973] to value the flexibility of the architecture to change. In this section, we formulate the ArchOptions model as expressed in (4.4) and explore in depth the analogy ArchOptions make with Black and Scholes. In the next sections, we interpret ArchOptions in the context of architectural stability and discuss related valuation issues and assumptions.

Let us assume that the architecture potential of a given system is V. As the software evolves, a change in future requirement i_i is assumed to buy x_i% of the architectural potential with a follow-up investment cost of C_{ei}, where C_{ei} corresponds to an estimate of the likely cost to accommodate the change. This is similar to a call option to buy (x_i%) of the base project, paying C_{ei} as exercise price. The investment

opportunity in the system can be viewed as an upfront investment, denoted by V_{Dev} plus call options on future opportunities, where a future opportunity corresponds to the investment to accommodate some future requirement(s). The payoff of the constructed call option gives an indication of how valuable the flexibility of an architecture to endure likely changes in requirements. The value of an architecture with a given system materializes to ArchOptions expressed in (4.4) and accounting for V_{Dev} and both the expected value and exercise cost of accommodating likely changes in requirements i_i, for $i \leq n$. ArchOptions is derived by mapping the economic characteristics of the architecture (under development or evolution) onto the parameters of the option model of (4.1) - as shown in Table 4.1. The economic characteristics include the development (evolution) effort, schedule, and budget. We assume that the risk-free interest rate is zero for the simplicity of exposition. We then pursue (4.2) and (4.3) to valuation which we explore in next sections.

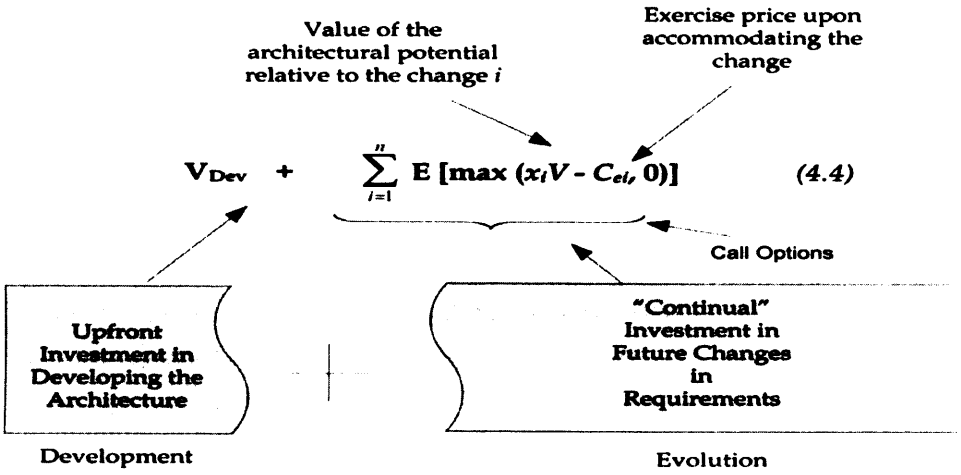


Figure 4.4. The ArchOptions model

Table 4.1. Financial/real options/ArchOptions analogy

Option on stock	Real option on a project	ArchOptions
Stock price(S)	Value of the expected cash flows	Value of the "architectural potential" in addressing a change in requirements($x_i V$)
Exercise price(X)	Investment cost	Estimate of the likely cost to accommodate the change (C_{ei})
Time-to-expiration(T)	Time until opportunity disappears	Time indicating the decision to implement the change (T)
Volatility(σ)	Uncertainty of the project value	"Fluctuation" in the return of value of $x_i V$ over a specified period of time (σ)
Risk-free interest rate(r)	Risk-free interest rate	Risk-free interest rate relative to budget and schedule (r)

Stock price = $x_i V$

In traditional applications, the real option analogy of stock price, S, corresponds to the value of the cash flows of the investment in a particular project. In ArchOptions, the S analogy corresponds to the value of the "architectural potential" in accommodating the change. In this context, we consider the architecture as a portfolio of assets (rather than a single asset). More specifically, we view the architecture as a portfolio of requirements. We argue that the value of the architecture is in the value of the requirements it supports during the software system operation or tend to support as it evolves. In ArchOptions, the nature of the change and the case determines the dimensions on which the value of the architectural potential is to be realized. Let us return to the motivating example we have highlighted in Section 4.2: the value of the architectural potential of inducing an architecture with J2EE and not CORBA (and vice versa) is a relative value. This value could span different dimensions including ease of future maintenance and relative savings in deployment and configuration of the software system, if we choose to go for J2EE and not a CORBA-induced architecture (and vice versa). This value is realized, if the likely change in future load materializes, necessitating scaling the system of the given architecture. Moreover, as we will see in Chapter 6, scalability is often measured by the *throughput* or the capacity of the system. Throughput is a generic performance criterion, which expresses the amount of work performed by the system during a unit of time. This criterion is based on the observation that for a

fixed system with a given throughput, there is an inverse relationship between the response time and the number of clients. In other words, the more requests clients submit, the longer are the delays. The value potentials of inducing the architecture with either CORBA or J2EE in response to change in load can hence be measured relative to throughput. In case of reengineering or designing for change, as it is the case when restructuring or refactoring the architecture of a legacy system, the value added is determined by the architectural potential realized by reengineering the architecture versus not exercising the reengineering decision. Again, the realized value may span several dimensions, such as ease of future maintainability, extensibility, modularity, reusability, complexity, and efficiency. Alternatively, the architecture could "pull" options by responding to changes in the market conditions, either with minimal changes to the architecture, by leaving the architecture of the software system intact, and/or by adapting new features and requirements with shorter time-to-market and gaining a competitive opportunity. In this context, the architectural potential relative to the change could have potential market value. Product-line architectures fit under this category as it could be argued that instantiating from the core architecture a new product is a trend towards "planned" evolution in accommodating variability in requirements across products, while respecting existing commonality, eventually with shorter time-to-market to gain a competitive market opportunity.

Exercise price = C_{ei}

The real option analogy of the exercise price corresponds to the investment cost in realizing the said change. The nature of the case determines the dimensions on which the cost needs to be assessed. Back to our motivating example, we can see that the cost of realizing a scalability change could differ from one version to another (i.e., the J2EE-induced or the CORBA-induced architecture) and with the architectural mechanism that is responsible in accommodating the change. Let us suppose that we take replication, as an architectural mechanism, to realize the load change. Obviously, the J2EE induced architecture has embedded options due to the built-in replication primitives. However, this flexibility comes with a cost, mostly on the licensing dimension. As for the CORBA induced architecture, the middleware needs to be modified and extra-functionalities need to implemented to realize scalability.

However, what is the exercise price that an enterprise need to pay if the system needs to scale to a high load in either structure? In general terms, the exercise price, corresponds to the cost of realizing scalability on each structure, given by C_{ei} for requirement *i*. As the replicas may need to be run on different hosts, calculating C_e as a function of the number of hosts, can be given by:

$$C_{ei} = \sum_{h=1...k} (C_{dev}, C_{config}, C_{deploy}, C_{licesh}, C_{hardw})_h, \qquad (4.5)$$

where, *h* corresponds to the number of hosts. C_{dev}, C_{config}, *and* C_{deploy}, respectively corresponds to the cost of development(if any), configuration, and deployment for the replica on host *h*. C_{licesh} and C_{hardw} respectively correspond to licenses and hardware costs, if any. All costs could be given in ($). Interested reader may refer to Chapter 6 for a detailed case study, where we show how these parameters are estimated on each structure.

In case of reengineering an architecture to facilitate future changes, as it is the case of refactoring, the investment in reengineering may create future options. This is because refactoring adapts flexibility into the architecture, making it more adaptable to changes. The option is said to be exercised and benefits may be realized only if some future changes need to be implemented on the given structure. The enterprise still needs to pay a cost for implementing the change; however, this cost could be relatively less expensive than the unrefactored structure. The cost could be measured in terms of man-months and could be cast into a monetary value.

Estimating cost is a well-researched field in software engineering; it is outside the scope of our work. In Chapter 5, we use well-established ways for estimating cost in software engineering, ranging from coarse-grained to fine-grained and parametric versus knowledge based.

Volatility = σ

Volatility is a quantitative expression of risk. Volatility is often measured by standard deviation of the rate of return on an asset price S (i.e., $x_i V$) over time. Unlike with financial options, in real options the volatility of the underlying asset's value cannot be observed and must be estimated. During the evaluation of architectural stability, it is anticipated and even expected that stakeholders might undervalue or overvalue the architectural potential relative $x_i V$ to the change in requirement(s). In other words, stakeholders tend to be uncertain about such value. For example, back to the motivating example of Section 4, suppose that the value of the architectural potential of inducing an architecture with J2EE and not CORBA (or perhaps vice versa) take the form of relative savings in development and configuration effort, if the future change in scalability need to be exercised on the induced structure: estimating such savings may vary from one architect to another within the firm. It differs with the architect's experience, the novelty of the situation; consequently, it could be overvalued or undervalued. The variation in the future savings, hence, determines the "cone of uncertainty" in the future value of the architectural potential for embarking on a J2EE-induced architecture relative to the CORBA one. Thus, it is reasonable to consider the uncertainty of the architectural potential to correspond to the volatility of the stock price. In short, the volatility σ tends to provide a measure of how uncertain the stakeholders are about the value of the architectural potential relative to change; it tends to measure fluctuation in the said value. In Chapter 5, we explore ways for estimating σ for our case.

Risk-free interest rate = r

The risk-free rate is a theoretical interest rate at which an investment may earn interest without incurring any risk. An increase in the risk-free interest rate leads to an increase in the value of the option. Finding the correspondence of this parameter is not straightforward, for the concept of interest in the architectural context does not hold strongly (as it is the case in the financial world) and is situation dependent. In our analogy, we set the risk-free interest rate to zero assuming that value of the architectural potential is not affected by factors that could lead to either earning or depreciation in interest. That is, the value of architectural potential today is that of the time of exercising the flexibility option. However, we note that it is still possible

for the analyst to account for this value, when applicable. For example, if the architectural platform is correlated in a way with the market, then the value of the architectural potential may increase or decrease with the market performance of the said platform. Similarly, suppose that development revolving around the said platform might be using external resources to maintain- such as, extra developers, money and/or tools borrowed from other projects- or might go beyond the assigned schedule and out of budget (which is the norm in software development), then the architecture is anticipated not to record any credits in interest, but rather a value deprecation. In these cases, the free-risk interest rate can be estimated relative to the budget and schedule.

Exercise time = T

The real option analogy of the exercise time (also referred to as time-to-expiration) corresponds to the time until the investment opportunity disappears. The time that the likely change(s) need to be exercised on the software system of the given architecture correspond to the time to expiration of the option. Back to the motivating example, the built-in replication primitives of J2EE continues to constitute an "unutilized" opportunity for future investment in scaling the software system to attain some future business benefits. Such an opportunity continues to hold until the enterprise wishes to scale up the software system, say as a result of a sudden increase in users, as it is the case of successful e-commerce systems. Alternatively, the time of exercising the options might correspond to a milestone in the enterprise strategic roadmap towards expansion of its services to new customer segments. The exercise time might also be coined with the lifespan of the general technologies on which the architecture is built (e.g., 20 years for databases, 10 years for middleware, and 2 years for user interface toolkits). That is, the change might be attributed to the "decay" or related "upgrades" in the exploited database, the underlying middleware, or the interface toolkits. Throughout the thesis, we use fictitious numbers for the exercise time.

Interpretation

For a likely change in requirement k, we interpret *(4.4)*:

The option is in-the- money

If $x_k V$ exceeds the exercise cost (i.e., E [max ($x_k V$ - C_{ek}, 0)]>0), then the flexibility of the architecture relative to the change is likely to payoff if the change is exercised. This means that the architecture is said to be *potentially stable* with respect to k. The more the option is in the money, the more valuable is the embedded flexibility; hence, the better are the potentials for the architecture to be stable relative to the change. In real situations, the architect/analyst is interested in selecting an architecture that maximizes the yield in options relative to some likely changes. An optimal selection could be when the option value approaches the maximum, indicating an optimal payoff in an investment in flexibility. The analyst may perform sensitivity analysis and analyze when such a situation is likely to occur. Returning to our running example, as we will see in Chapter 6, upon calculating the call options for the change in scalability on the J2EE-induced architecture, S_1, relative to that of the CORBA-induced architecture, S_0, the options are said to be in-the-money for S_1. In particular, ArchOptions shows that S_1 is in the money relative to the development, configuration, and the deployment, if the change in scalability need to be exercised in one year time. It is worth pointing out that though S_1 is flexible relative to the scalability change, it might not necessarily mean that it might be flexible with respect to other changes. Obviously, J2EE provide the primitives for scaling the software system, which result in making the architecture of the software system more flexible in accommodating the change in scalability, as when compared to the CORBA version. As we will see in Chapter 6, the structural analysis has completed the option analysis to verify the stability of S_1 relative to the change and to quantify the impact of the change on the architecture. The intuition is that complementing the structural impact analysis with a value-based back-of-the-envelope calculation, the combination provides the architect/analyst with a useful tool for understanding the extent to which the software system tends to be flexible relative to a likely change in requirements, a cost/value indictors of the impact of

the change on the structure, the likely success (failure) of the software system evolution, and consequently the potential stability of the software architecture relative to the change.

The option is out -of- money

If the value of the call option sinks to zero (i.e., E [max ($x_k V$ - C_{ek}, 0)]>0), then the flexibility of the architecture in response to the change is not likely to add a value. Two interpretations might be possible:

(i) The architecture is overly flexible in the sense that its response to the change has not "pulled" the options. This implies that the embedded flexibility (or the resources invested in implementing flexibility) are wasted and unutilized to reveal the options relative to *this* change. In other words, the degree of flexibility provided is much more than the flexibility demanded for this change. This case has the prospect in providing an insight on how much do we need to invest in flexibility to achieve stability relative to the likely future changes, while not sacrificing much of the resources. In Chapter 6, the refactoring case provides a good example to illustrate this. We will see that by refactoring the original structure, we have obtained a more flexible one that has better prospect of accommodating the change. Though S_1 is flexible, refactoring has not "pulled" the options for one change. The refactored structure is reported to be out of the money for one change. This implies that the embedded flexibility (or the resources invested in implementing flexibility) is wasted and unutilized to reveal the options relative to one change. In other words, the degree of flexibility provided is much more than the flexibility demanded for this change. We have repeated the experiment, but stressing refactored structure with two, three, four, and then five average changes at a time. Using two average likely changes, the options reported zero values. Again, two likely average changes have not "pulled" the options. Interestingly, the refactored structure was just about to pull the options for three changes, as we will see in Chapter 6. For four, five, and nine changes, the structure has revealed the options.

(ii) The other case is when the architecture is inflexible relative to the change. This is when the cost of accommodating the change is much more than the cumulative expected value of the architecture responsiveness. Returning to our running example, as we will see in Chapter 6, calculating the options on the CORBA-induced architecture S_0, relative to that of the J2EE-induced architecture S_1, we can see that S_0 is said to be out of the money for the scalability change. The CORBA version has not added value, relative to J2EE, as the cost of implementing the services responsible for realizing the change in scalability was relatively significant to "pull" the options. As we will see in Chapter 6, the structural analysis has completed the option analysis to verify the instability of S_0 relative to the change in scalability.

Valuation Issues and Assumptions

In this subsection, we clarify some theoretical issues revolving around the valuation of ArchOptions(4.4) and on estimating its parameters. The options model (4.4) requires the estimation of several parameters. Most importantly are C_{ei}, x_iV, and σ which respectively correspond to exercise cost of implementing the i^{th} change in the system of the given architecture, the value of the architectural potential relative to the i^{th} change, and the fluctuation of this value. Below, we briefly show how these parameters could be estimated. In Chapter 5, we provide in depth treatment to the estimation of the ArchOption's parameters and inline with the proposed method.

The derived ArchOtions model is a general real-options model; it could be valued using existing techniques to options valuation. We adopt model (4.3) of Black and Scholes to the valuation of the constructed call options. Alternatively, we could have cast the options model to use different options valuations (e.g., [Cox et al., 1979]). However, the application of [Black and Scholes, 1973] offers a closed and an easy-to-compute solution, for it assumes that x_iV is lognormaly distributed, not requiring x_iV to be probability-adjusted for rise and drop in value, as when compared to [Cox et al., 1979]. We note that it remains an open challenge to strongly justify precise estimates for real options in software [Sullivan et al., 1999]. Following the argument

of [Sullivan et al., 2001], such models need not be perfect: what is essential is that they capture the most important terms; their assumptions and operation must be known and understood so that the analyst can evaluate their predictions. Experts may question our use of Black and Scholes [1973] to options valuation, as the satisfaction of the spanning condition may be doubtful. Real options may be valued similarly to financial options, though they are not traded [Schwartz and Trigeorgis 2000]. For a change in requirements, the call E [max ($x_i V - C_{ei}$, 0)] (4.6) at expiration is valued using the above (4.2) and (4.3) of Black and Scholes and detailed as follows:

$$E\,[\max (x_i V - C_{ei}, 0)] \qquad (4.6)$$

$$C = x_i V\, N(d_1) - C_{ei} e^{-r(T)}\, N(d_2)$$

where,

$$d_1 = \frac{\ln(x_i V / C_{ei}) + (r+\sigma^2/2)(T)}{\sigma(T)^{1/2}}$$

$$d_2 = \frac{\ln(x_i V / C_{ei}) + (r-\sigma^2/2)(T)}{\sigma(T)^{1/2}} = d_1 - \sigma(T)^{1/2}$$

Finding a twin asset

Real options valuation based on Black and Scholes pricing technique determines the value of an asset in question in span of the market value using a correlated *twin asset* [Schwartz and Trigeorgis 2000]. The twin asset is an asset that has the same risks as the asset in question will have when the investment has been completed [Schwartz and Trigeorgis 2000]. The intuition is that to understand the behavior of the asset in question, we can use a twin asset, also referred to as a replicated portfolio. The assumption is that under similar conditions the twin asset and the asset in question are interchangeable for all practical purposes and should be worth the same. That is, if we know how much the twin asset is worth in the present, we can then determine how much the option on the asset in question is worth in the present.

Software architectures, however, are (non-traded) real assets. Real options may be valued similarly to financial options, though they are not traded [Schwartz and Trigeorgis 2000]. To facilitate valuation using the principle of a twin asset, we consider the architecture as a portfolio of assets (rather than a single asset). More specifically, we view the architecture as a portfolio of requirements. In this context, we argue that the value of the architecture is in the value of the requirements it supports during the software system operation or tend to support as it evolves. This assumption facilitates valuing the architectural potential in supporting the change based on a similar experience. It can also help in calibrating the architectural potential in supporting the changes with the business or the market value, when available. Consequently, valuing the architectural potential to the change requires finding a twin asset with similar characteristics to the one at hand. We argue that reusing a past development experience such as previous design and its corresponding implementation to inform the valuation bear a resemblance to the concept of a twin asset. We also argue that much of the valuation effort in software engineering is based on person-months. Such valuation does implicitly hold market-based data and is still done in relation with the market and based on similar experience. Back to our motivating example, in chapter 6, we can see that in valuing the architectural potential of the CORBA-induced version relative to that of J2EE, we have used a previous design and development experience, where the scalability change has been designed and implemented on a CORBA complaint middleware, TAO (refer to Chapter 6). In this context, we argue that our use for the design and the corresponding implementation of scalability on TAO as guidelines bears a resemblance to the concept of a twin asset, for we are reusing a past development experience to inform the valuation. In Chapter 6, we will also see how using published performance benchmarks to value the architectural potential, relative to likely changes in scalability requirements, resemble the twin asset.

Estimating $x_i V$

In financial options, several proxies are available to predict the value of the financial asset - the most obvious proxy is simply the historical values of the asset. In real options, such proxies rarely exist and the analyst may need to rely on experience and judgment in her/his estimations [Schwartz and Trigeorgis, 2000]. Real options

valuation focuses on market value and uses the return on the twin asset as an input to the valuation of the asset in question. If the asset value is not directly observable, it is reasonable to use estimates of the revenues on the asset to estimate the market value [Schwartz and Trigeorgis, 2000].

The architectural potential relative to the changes in requirement can be valued in terms of the directly observable cash flows linked to future operational benefits or the market value, making it easy to use the return on the twin asset to value the options. In many others cases, the architectural potential may not be directly observable through cash flows; the analyst(s) may then need to rely on experience for estimation. If the analyst relies on experience and judgment in her/his estimation, the estimates tend to be subjective but could make an implicit use of market information. However, back-of-the-envelope calculations, which are based on value estimates (rather than on market value), are yet informative [Sullivan et al., 2001].

As a compromise, we argue the valuation of x_iV is a multi-perspective valuation problem. That is, valuing the architectural potential to the change necessarily requires a comprehensive solution that is flexible to incorporate multiple valuation techniques; some with subjective estimates and others based on market data, when available. The problem of how to guide valuation and introduce discipline in this setting, we term as the *multiple perspectives valuation problem*. To address this problem, Chapter 5 outlines a conceptual *valuation points of view* framework. The framework aims at capturing and valuing the flexibility of the architecture to the change from different perspectives. In Chapter 6, we exemplify the use of the framework for capturing the options from different perspectives.

Estimating σ

The volatility of the stock price (σ) is a statistical measure of the stock price fluctuation over a specific period of time; it is a measure of how uncertain we are about the future of the stock price movements. Schwartz and Trigeorgis [2000] describe three possible ways for calculating the volatility. The first way is to make an educated guess. One approach is to examine a range of estimates from say 30% to 60% and guess which might be the most appropriate. A second approach is to gather

historical data on investment returns in the same or related industries. Another approach is to simulate. Projections of a project's future cash flow, together with Monte Carlo simulation techniques, for example, can be used to synthesize a probability distribution for project returns and from this σ can be calculated.

The application of Black and Scholes [1973] assumes that the stock option is a function of the stochastic variables underlying stock's price and time. In ArchOptions, volatility stands for the "fluctuation" in the value of the estimated *xiVs*. Intuitively, it "aggregates" the "potential" values of the structure in response to the change(s). In Chapter 5, we explore ways for estimating volatility inline with the method. In some cases, we take modeling assumptions for volatility and based at the information at hand. In other cases, we assume that value (x_iV) moves stochastically bounded to two extreme values: optimistic and pessimistic. This assumption appears to be plausible: (i) it tends to account for all possible values within the bound, yielding to a better approximation when opposed to an ad-hoc type of estimation; (ii) the value of an (evolvable) architecture changes over time; it tends to change in uncertain way due to changes in requirements. We estimate variation on these values, explained in Chapter 5. We use the standard deviation of the variation of the three *xiVs* estimates-the optimistic, likely, and pessimistic values, to calculate σ and adhering with the real options principles to the valuation of σ.

Estimating C_{ei}

As we mentioned before, cost estimation is a well-researched component in software engineering; it is outside the scope of our work. For example, it is feasible to use existing metrics to cost estimation (e.g., COCOMO-II [Boehm et al., 1995]). This is due to the fact that a considerable part of the distributed applications implementation could be already available, when the architecture is defined, for example, during the Elaboration phase of the Unified Process. Another approach is to build on architectural level dependency analysis (e.g., [Stafford and Wolf, 2001]) research to extract cost estimates of accommodating i_i, guided by some structural criteria.

Generally speaking, ArchOptions is flexible to incorporate either coarse-grained or fine-grained estimation of the cost of implementing the change in the model. Generally, two extreme routes can be pursued for estimating the cost of the change in software engineering: expert knowledge or parametric models to cost estimation. When expert knowledge is combined with parametric knowledge, more precise estimation are said to be realized. Note that the granularity of the estimation is dependent on the case and the information available for the evaluation. In the next Chapter, we sufficiently address how the cost could be estimated using parametric-models and/or expert knowledge.

Sensitivity Analysis

Statistical questions on how the uncertainty of the input parameters propagates to the model output often require sensitivity analysis. The objective is to provide an understanding of how the model responds to changes in input parameters. For example, the estimated parameters may be subject to uncertainty: parameters values could have been overestimated or underestimated. Further, the estimated value may be liable to further adjustment to reflect the time value. We support the model with sensitivity analysis to increase the confidence in the model predictions and to provide a basis for "what-if" analyses.

First derivative analysis is much used in the investment arena for analyzing the sensitivity of the value of a financial option to changes in the variables. *Delta* and *Vega* provide the investment analyst with a ready means to discover financial option's sensitivity to changes in the estimated value of the underlying asset; and increases and decreases to the volatility of the underlying asset.

Table 4.2 provides a summary of the sensitivity parameters, their financial explanation, mathematical formulation and the corresponding ArchOptions analogy.

Table 4.2. Sensitivity parameters and ArchOptions

Parameter	Financial Explanation	ArchOptions Analogy	Math-formula
Delta (Δ)	Option price rate of change w.r.t. the underlying asset (%)	Option value rate of change w.r.t. x_iV	$\frac{\partial c}{\partial(x_iV)}$
Vega (ν)	Option price rate of change w.r.t. the volatility of the underlying asset (%)	Option price rate of change w.r.t. σ (%)	$\frac{\partial c}{\partial \sigma}$

The *Delta (Δ)* of an option is defined as the rate of change of the option price with respect to the underlying asset. Suppose that the delta of a call option is 0.6. This means that when the underlying asset price changes by a small amount, the option price change by about 60% of that amount. Mathematically, delta is the partial derivative of the call price with respect to the underlying asset price given by $\Delta = \partial C/\partial S$. In practice, volatilities may change over time. This means that the value of the option is liable to change because of the movement in volatility as well as because of changes in the asset price and the passage of time. The *Vega (ν)* of an option is the rate of change of the value of the option with respect to the volatility of the underlying asset. If Vega is high, the option value is very sensitive to small changes in volatility. If Vega is low, volatility changes have relatively little impact on the value of the option.

4.4 Uses

ArchOptions could provide a basis for analyzing many architecture-centric evolution problems, which place considerable emphasis on the flexibility of the architecture to ease software evolution. The model can provide insights into the economics of flexibility, the inflexibility, and the over-flexibility of the architecture and its associated artifacts relative to the change. In this context, the model intends at answering the following key question: how much worth is it "buying" flexibility to facilitate future changes and support the development (evolution) of potentially stable architectures? The model has the prospect of valuing the architectural flexibility to various types of changes. These could be functional or non-functional.

These changes could be preventive, adaptive, or perfective [IEEE Standard 610.12, 1993], with the assumption that the architecture guides the evaluation. For example, preventive and perfective types of changes may aim at introducing further flexibility into the architecture of the software system or its associated artifacts. For these changes, the model provides the analyst/architect with a mean to value the worthiness of investing in an architectural design decisions, which adapts flexibility to facilitate future growth.

ArchOptions may aim at providing the analyst/architect with insights into architectural stability and investment decisions related to the evolution of a software architecture. In ArchOptions, the value of the constructed calls are indicative measures of the "architectural potential" in unlocking future growth opportunities (e.g., case of reuse, new market products), enhancing the upside potentials of the architecture, generating value (e.g., savings in maintenance), or incurring loses (e.g., case of a disruptive changes), as a consequence of accommodating the change. The value of the calls may assist the analyst/architect in strategic *"what if"* analyses, to inform:

- the worthiness of designing or reengineering the architecture for change;
- the retiring and replacement decisions of either the architecture or its associated design artifacts;
- the decisions of selecting an architecture, architectural style, middleware, and/or design with desired stability requirements;
- the trade-off between the upfront cost of enabling the change on the architecture of the software system and the long-term future benefits as a result;
- the compromise between the architectural "intactness" and the cost-effectiveness of amending the architecture to accommodate the change;
- the trade-offs between two or more candidate software architectures for stability and the value added;

- the strategic position of the enterprise- if the enterprise is highly centered on the software architecture (e.g., the case in web-based service providers companies);
- and/or the success (failure) of evolution.

Apart from the above architecture-centric evolution problems, it could be argued that the incremental software processes, such as the unified process, are also ways to structure the software's evolution through prescribed steps [Jazayeri, 2000]. The assumption is that evolution is helped by the feedback gained from releases of the early increments. The construction of the first release of the system is only the first of many milestones in this evolution [Jazayeri, 2000]. In the context of applying ArchOptions, an iterative and intertwined phased development (process) is flexible to allow the change in requirements to be exercised at the end of each iteration (phase) to mitigate risks before proceeding to a next iteration (phase) and render a more stable architecture. For instance, under RUP, the Life-Cycle Architectural (LCA) milestone corresponds to the time where the detailed system objectives and scope are examined, the choice of the architecture is (re) considered, and the major risks are identified. Accordingly, the LCA could be the time where the options are constructed and their payoffs are predicted- if exercised at a time in the future. In the case of an iterative and intertwined development (evolution) process, the time to expiration corresponds to the estimated time to deploy a successful software generation. In the evolution context, a successful software generation is assumed to have the change in requirements accommodated by that time.

In Chapter 6, we will explore how the ArchOptions model could be applied to reason about two architecture-centric approaches to evolution. These are (i) valuing the payoff of re-engineering the structure of the software system to facilitate future changes in requirements and (ii) informing the selection of a more stable middleware-induced software architecture, relative to future changes in scalability. In Chapter 7, we will highlight some possible unexplored uses of the model to reason about the worthwhile of investing in restructuring of systems to support aspect-orientation, with the objective of facilitating future maintainability and better stability.

ArchOptions could benefit from tool support. The envisioned tool may automate the model, provide basis for estimating its input parameters, and tailor the output based on the objective of applying the model. The tool may automate or provide a support for much of the activities to be discussed in Chapter 5. The tool may combine spreadsheet capabilities to computation and visualization of the results with mining of software repositories for storing, maintaining, and analyzing project's versions and potential twin assets.

4.5 Related Work

In this subsection, we provide a quick overview of closely related research on the use of real options in software design and engineering. The use of real options has taken two forms: (i) quantifying investments in software in relation to the market and (ii) understanding the nature, role, and value in options with the objective of linking structural design and engineering to value. The latter category aims at addressing core issues in design and engineering of software by linking technical engineering issues to value creation. We scope the review on this category, as our use of real options theory fits under it.

Economics approaches to software design appeal to the concept of static Net Present Value (NPV) as a mechanism for estimating value [Boehm and Sullivan, 2000]. These techniques, however, are not readily suitable for strategic reasoning of software development as they fail to account for flexibility [Boehm and Sullivan, 2000; Erdogmus et al., 1999]. The use of strategic flexibility to value software design decisions has been explored in, for example, [Erdogmus and Vandergraff, 1999; Erdogmus and Favaro, 2002; Erdogmus 2000; Sullivan; 1996; Sullivan et al., 1999; Sullivan 2001] and real options theory has been adopted to value the strategic flexibility:

Baldwin and Clark [1993; 2001] pioneered the use of real options in systems design and engineering. They were the first to study the flexibility created by modularity in design of components (of computer hardware systems) connected through standard

interfaces. Their theory accounts for the influence of modularity on the evolution of computer system designs and the structure of the industry that creates them. In particular, Baldwin and Clark's theory is based on the idea that modularity (in computer systems) adds value in the form of *real options*. They consider that modularity in design multiplies and decentralizes *real options* that increase the value of a design. A monolithic system can be replaced only as a whole. That is, there is only one option to replace, and exercising it requires that both the good and the bad parts of the new system be accepted. In a sense, the designer has one option on a portfolio of assets. A key result in modern finance, however, shows that all else remaining equal, a portfolio of options is worth more than an option on a portfolio. In contrast, in ArchOptions consider the architecture as portfolio of options, where the options are held on the architectural potential in supporting the change in requirements.

Baldwin and Clark's method has two main components, the Design Structure Matrix (DSM) and the Net Option Value formula (NOV). DSM represents the design of a system by a structure matrix, providing an intuitive, qualitative framework for design. Example of a DSM is depicted in Figure 4.5. The rows and columns of a DSM are labeled by the design parameters. A dependency between two design parameters is represented by a mark (X). A mark in row B, column A means that an efficacious choice for B depends on the choice for A. NOV quantifies the consequences of a particular design, thus permitting a precise comparison of differing designs of the same system. NOV reasons about the value added to a base system by modularity upon applying a modular operator. Module operators include *substitution*, which substitute a modular with an alternative, *augmentation*, which adds a module to a system, *exclusion*, which removes a module, *inversion*, which standardizes a common design element, and *porting*, which transports a module for use in another system. The NOV model answers the following key question: "How much is it worth to be able to substitute, augment, exclude, invert, or port modules?" For example, the NOV for quantifying the options added as a result of substitution uses the following reasoning: A module creates an opportunity to invest in k experiments to (a) create candidate replacements, (b) each at a cost related to the complexity of the module, and, (c) if any of the results are better than the existing choice, to substitute in the best of them, (d) at a cost that related to the visibility of the module to other modules

in the system. Baldwin and Clark acknowledge that designing modularizations is not free; but, once done, the costs are amortized over future evolution. The NOV model ignores those costs, though accounting for them is important.

	A	B	C
A	.		
B	X	.	X
C		X	.

Figure 4.5. Example of a Design Structure Matrix (DSM) [Baldwin and Clark, 2001]

Sullivan et al. [1996; 1999; 2001] pioneered the use of real options in software engineering. Sullivan et al. [1996; 1999] suggested that real options analysis can provide insights concerning modularity, phased projects structures, delaying of decisions and other dynamic software design strategies. Sullivan et al. [1999] outline an options-based interpretation of the spiral-model for software development. Sullivan et al. [1999] view that the spiral-model provides flexibility in at least two important dimensions. First, it imposes a phased structure on a project, where the goal of each phase is to reduce a key uncertainty facing the project, with decisions about whether or how to invest in subsequent phases based on information from earlier phases. Second, within each phase it stresses the development of alternatives, creating an option to pick the most promising one. In context of real options, Sullivan et al. appeal to the use of options to defer decisions to invest until optimal to do so [Dixit and Pindyck, 1994; Madj and Pindyck, 1987; Myers, 1977] and option to explore from a development alternative mainly for mitigating risks upon selecting a risky asset [Stulz, 1982].

Sullivan et al. [1999] approach to options pricing uses events trees. They note that the first step for software engineering is to understand the nature and role of options. They added that the next step is to develop option models. Sullivan et al. [1999] address the first step. In contrast, our work covers both steps: we seek an understanding for the architectural stability problem from an options perspective (see Chapter 6). We develop a model that complements such an understanding. Sullivan et al. [1999] formalized that option-based analysis, focusing in particular on the flexibility to delay decision making. In particular, they addressed the timing of

design decisions, where they discussed the role of options in decisions about time-to-market under threat of competitive entry, and the engineering tradeoffs that are appropriate in such circumstances. In contrast, ArchOptions is concerned about the *growth options* an architecture can provide in the face of uncertainty attributed to change.

Sullivan et al. [2001] extended Baldwin and Clark's theory [2001] that is developed to account for the influence of modularity on the evolution of the computer industry(sufficiently described above). Sullivan et al. [2001] use the model developed in [Baldwin and Clark, 2001] to treat the evolvability of software design using the value of strategic flexibility. Specifically, they argued that the structure and value of modularity in software design creates value in the form of real options. A module creates an option to invest in a search for a superior replacement and to replace the currently selected module with the best alternative discovered, or to keep the current one if it is still the best choice. The value of such an option is the value that could be realized by the optimal experiment-and-replace policy. Knowing this value can help a designer to reason about both investment in modularity and how much to spend searching for alternatives. Sullivan et al. [2001] apply Baldwin and Clark's substitution NOV model to compute quantitative values of the two modularizations, using parameter values derived from information in the DSM's combined with the judgments of a designer. The results are back-of-the-envelope predictions, not precise market valuations. Like in Baldwin and Clark, Sullivan et al.'s use of NOV ignores the costs of designing modularizations. They assume that once modularization is done, the costs are amortized over future evolution. Yet they acknowledge that accounting for the costs is important. In contrast, ArchOptions explicitly accounts for the cost of exercising the change on the structure of the system. It uses either parametric models or expert judgment for estimating the cost. When the cost, is an upfront cost for adapting flexibility into the system, ArchOptions adjusts the model to account for the upfront costs (see Chapter 6).

Erdogums [1999] describes how strategic flexibility in software development, involving COTS components, can be valued using real options. They apply two quantitative valuation methods, NPV and real options, to the assessment of the COTS-centric software development projects. The objective is to investigate the

economic incentive of choosing COTS centric strategy in a project vis à vis the alternative, the custom development. Real options is employed to investigate the value of strategic flexibility inherent in COTS-centric development. The analysis concentrates on the impact of the risk embedded in the COTS product and the development time. The result shows that real options theory is preferred over NPV analysis, as NPV ignores the value of the flexibility in COTS-centric projects making it appear less attractive.

Bergey et al. [2001] proposes the Options Analysis technique for Reengineering (OAR). OAR is a systematic, architecture-centric, decision-making method for identifying and mining software components within large, complex software systems. Mining involves rehabilitating parts of an old system for reuse. OAR identifies potentially relevant architectural components and analyzes the changes required to use them in a software product line or new software architecture. In essence, OAR provides a set of mining options along with estimates of the cost, effort, and risks associated with those options. OAR is motivated by the fact that existing components are often poorly structured and poorly documented and they differ in levels of granularity. There is no clear guidance on how to salvage components. OAR's five activities identify potential components, estimate the mining cost, and evaluate the effort required to reuse legacy components. OAR reveals implicit stakeholder assumptions, constraints, and other major drivers that affect component mining, thereby giving managers insight into this complex task. OAR aims at making the decisions required to cost-effectively and efficiently mine legacy system components.

An interesting use of real options theory is that of [Erdogums and Favaro, 2002]. Erdogmus and Favaro use real options to value the inherent flexibility in Extreme Programming (XP), where they have considered XP as a lightweight process that is well positioned to respond to change and future opportunities; hence, creating more value than a heavy-duty process that tends to freeze development decisions. They use real options to reason about one of the most widely publicized principles of XP, the *You Aren't Going to Need It* principle (YAGNI). The YAGNI principle highlights the value of delaying an investment decision in the face of uncertainty about the return on the investment. In the context of XP, this implies delaying the

implementation of fuzzy features until uncertainty about their value is resolved. YAGNI is a typical example of *option to delay*. Erdogmus and Favaro observed that the delay option underlying the YAGNI scenario is much akin to a financial options. Their results reveal that under increasing future cost assumptions to the implementations of the features, waiting does not make economic sense. This is because delaying the implementation decision destroys value because the increase in the cost of change overtakes the benefit of the flexibility to make the implementation decision later. As a result, the longer we wait, the less value we create. When uncertainty is high or it is expected to be resolved over the long term, decisions about system features should be committed to as late as possible; otherwise, they should be committed to now. Finally, under a constant cost function, commitment should always be made later rather than sooner. Hence, Erdogmus and Favaro uses real options theory to reason about the *option to delay* implementing features in relation to XP. In contrast, ArchOptions is concerned about the *growth options* an architecture can provide in the face of uncertainty attributed to change.

Plausible improvements of the existing Cost Benefit Analysis Method (CBAM) [Kazman et al., 2001], sufficiently described in Chapter 2 of the thesis, include the adoption of real options theory to reason about the value of postponing an investment decisions in an architectural strategy. In the situation where many architectural strategies are considered, CBAM attempted to apply real options theory based upon the dependency structure of the strategies. For example, let AS2 and AS3 be two architectural strategies, where AS2 is low-cost, low-benefit, and AS3 is high-cost, high benefit. Analysis of the dependency structure may show, for example, that AS2 must be first be implemented, deferring the implementation of AS3. In other word, CBAM uses real options theory to calculate the value of *option* to *defer* or *delay* the investment into an architectural strategy (i.e. the options to defer the investment until more information will be available).

As we have noticed from the above overview on related work, work on real options has mainly focused on two types of options. These are the *options to explore* and *options to delay*. The objective is to reason about core issues in software and design in relation to timing as a way for treating uncertainty. In contrast, we have looked at a special category of options, which is referred to as *growth options*. As we mentioned

before, growth options are often embedded in platform-based applications. We use real options to predict architectural stability in the face of likely evolutionary changes in requirements. We value flexibility of the architecture to expand in the face of these changes; henceforth, what we value are the created growth options. For likely evolutionary change(s), we construct call options to value the flexibility of the architecture to accommodate the change(s). The value of the constructed calls are indicators of the ability of an architecture to unlock future growth opportunities and enhance the upside potentials of the architecture. Knowing this value can assist in predicting architectural stability.

It is worth noting that the use of economic models to assess the cost and value of software requirements have been explored, for example, in [Karlsson et al., 1997; Karlsson and Ryan, 1997; Sivzattian and Nuseibeh, 2001].

Karlsson and Ryan [1997] use a cost-value approach for prioritizing requirements. Karlsson and Ryan defined *requirements value* as the ability of a requirement to contribute to the customer satisfaction with the overall system, when successfully implemented. A *requirement's cost* is an estimate of the additional cost required to meet that requirements alone. By relating requirements value to its cost, stakeholders have a measure of that requirement's ability to contribute to customer satisfaction. Different stakeholders apply a ratio scale of intensity for pair-wise comparisons to assess the relative value/cost of candidate requirements. Analytical Hierarchy Process (AHP) [Saaty, 1980] is used to calculate each candidate requirement's relative value and cost of implementation. These are then plotted on a cost-value diagram that serves as a conceptual map for analyses, discussion, and prioritization.

Sivzattian and Nuseibeh [2001] propose a market-driven approach to supplement the prioritization and selection of requirements. Sivzattian and Nuseibeh argued that portfolio-based reasoning is well suited to inform the objective selection of requirements as it makes the connection between the selection decision and the market explicit. Unlike Karlsson and Ryan's approach, Sivzattian and Nuseibeh focus on the market value of the requirement and ignore the cost of corresponding implementation on the system, which is often crucial and must be considered in the prioritization process.

In contrast, in ArchOptions, the value is in the architectural potential in supporting a change in requirements. The cost corresponds to the cost of accommodating the change on the architecture of the software system and analogous to the cost of exercising an options. Karlsson and Ryan [1997] acknowledge that assessment of value and cost of implementation are based on decision makers' "experience and judgment that this could be supplemented by other methods". In our assessment, this could lead to a variation in possible value ascribed to the architecture in supporting the change. The volatility parameter of ArchOptions provides a closed solution for modeling such variation.

4.6 Summary

We have pursued an economics-driven approach to address the requirements for evaluating architectural stability. We have motivated the use of real options theory and have devised a real option model, referred to as ArchOptions, as a solution. We have described the approach taken, which is based on a simple and intuitive analogy with Black and Scholes[1973] options theory. We have reported on ArchOptions formulation, its possible interpretation, and its sensitivity. We have discussed valuation issues and assumptions under ArchOptions. We have highlighted possible uses of ArchOptions in analyzing many architecture-centric evolution problems. We have provided an overview of closely related work on the use of real options is software engineering.

Chapter 5

A Method for Applying ArchOptions

In the previous chapter, we have presented the ArchOptions model for predicting architectural stability. In this chapter, we support the model with a three-phase method for evaluating architectural stability. According to [Brinkkemper,1996], a software engineering method is *"is an approach to perform a system development project, based on a specific way of thinking, consisting of directions and rules, structured in a systematic way in development activities with corresponding development products"*. The method, which this Chapter describes, provides such directions and rules for applying the ArchOptions model by describing possible ways for estimating the model parameters. We describe phases for conducting an architectural evaluation for stability using ArchOptions. We discuss issues related to conducting these phases, as it was realized in its application (Chapter 6).

The method is structured in three phases. Figures 5.1 and 5.2 depict an overview of the method phases. In the first phase, the method assists in eliciting the likely changes in requirements. The method pursues scenarios to describe the likely future changes in requirements that are critical to the evaluation. In reality, a scenario could be further refined to correspond to one or more further changes that may need to be realized or could impact the architecture of the software system. To link the likely future change in the requirement to the architectural artifacts, Goal-Oriented Requirements Engineering (GORE) paradigm (e.g., [Dardenne et al., 1993; Anton, 1996]) could be adopted. The objectives are (i) to provide a paradigm, which traces the change in the requirement, exemplified by the scenario, into the architecture and

(ii) to analyze changes that are necessary to be made to accommodate the change, so we can quantify the flexibility of the software system in responding to the change.

In the second phase, we use a multi-perspective valuation points of view framework for valuing the flexibility of an architecture to change. The valuation using ArchOptions requires a comprehensive solution that incorporates multiple valuation techniques, some with subjective estimates, and others based on market data, when available. The solution shall be comprehensive enough to account for the economic ramifications of the change, its "global" impact on the architecture, and on other architectural qualities. We refer to the problem of how to guide the estimation in this setting as a *multiple perspectives valuation problem*. We describe the problem from a value-based software engineering perspective. To introduce discipline into this setting and capture the value from different perspectives, we use valuation points of view (e.g., market, structural, behavioral...) as a solution. The solution aims to promote comprehensiveness in accounting for the "global" impact of the change on one or more architectural quality. The solution also aims to promote flexibility through incorporating both subjective estimates and/or explicit market value, when available. For every valuation point of view, we construct call options for the given change. We estimate the cost of accommodating the change. This cost corresponds to the exercise price. We value the architectural potential in accommodating the change. The value of the architectural potential may take the form of future savings in maintainability, possible revenues due to the support of new services, new market products, and so forth. At the end of the second phase, the major inputs of the ArchOptions model would have been identified. In the third phase, we interpret the call values relative to the set evaluation objective.

Phase I. Eliciting and tracing the change to the architecture

Input:

An architecture and objective for evaluation

Process:

a) Set the objectives for evaluating architectural stability,

E.g., valuing the cost-effectiveness of designing/re-engineering for change, software architecture trade-off analysis, etc.

b) Elicit the changes $\{i_1, i_2, \ldots, i_n\}$ that are critical to the set objectives,

Case of planned changes:

E.g., Use technology roadmap and the roadmapping process to elicit the scenarios of planned changes

Case of extreme changes:

E.g., Use exploratory scenarios to check for extreme/unforeseen changes

c) Relate the change to the architecture

Identify goals from scenarios

E.g., Use heuristics and guidelines suggested by [Anton, 1997] to identify the goals

Trace the goals to the architecture

For each goal,

Refine the goal using knowledge of the solution domain until a trace is established with the associated architectural artifacts, which implement or said to be impacted by the change.

Output:

A systematic trace (structural) of the change in requirements to the associated architectural artifacts, which implement or said to be impacted by the change.

Figure 5.1. Phase I of the method

Phase II. Valuing the flexibility of the architecture relative to the change

Input:

An architecture (and its associated artifacts); objective for evaluation; systematic trace (structural) of the change in requirements to the associated architectural artifacts

Process:

Using the valuation objectives, identify the *valuation points of view*

For every *valuation point of view, P* Do

Construct *call options* to value the architectural flexibility relative to the change:

a) Calculate C_{ep}: Estimate the cost of the architectural strategy, mechanisms, and/or the associated implementations, which realize the change- the cost corresponds to the exercise price

E.g., Use expert knowledge to cost estimation Or Use parametric models to cost estimation (e.g., COCOMO II [Boehm et al., 1995]) Alternatively, Combine expert knowledge with parametric models.

b) Using the valuation objectives, identify the value of the architectural potential to the change:

I. Calculate $x_i Vp$: Using the set objectives for valuation, value the architectural potential to the change and relative to *this* point of view:

E.g., Limit the valuation to three: optimistic, likely, and pessimistic, or use valuation scenarios, etc.

II. Calculate σ_p:

E.g., Estimate the likely variation for the optimistic, likely, and pessimistic values or estimate the likely variation in valuation scenarios:

Compute the standard deviation of the elicited variations.

Alternatively, make a modeling assumption of σ or make an educated guess of σ_p.

Calculate the *call options* relative to change and the valuation points of view

Output:

Call options relative to the valuation point of view:

Cases where the call options: in-the-money and/or out-of-the-money

Figure 5.2. Phase II of the method

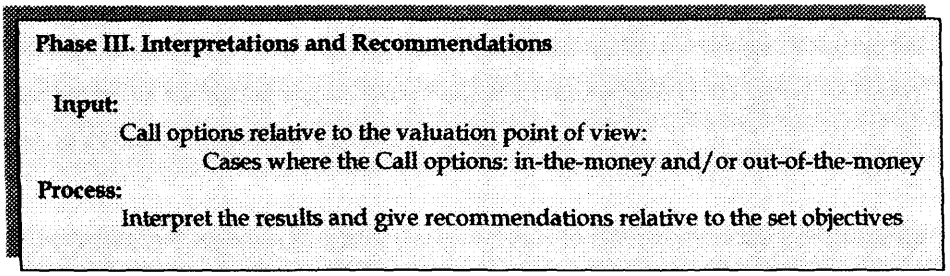


Figure 5.3. Phase III of the method

5.1 Phase I. Eliciting and Tracing the Change to the Architecture

Step I-a. Setting the objectives for evaluating architectural stability

For this step, the application of ArchOptions entails identifying the objectives that the stability evaluation needs to address. In the previous Chapter, we have highlighted several uses of the ArchOptions model for addressing some representative architecture-centric evolution problems. The objective for conducting an evaluation for architectural stability is often tailored to the said problem.

Understanding what drives the evaluation is essential for:

(i) identifying changes that are critical for analyzing the said objectives, which will be explored in this phase;

(ii) identifying both the value of the architectural potential relative to the change and the valuation dimension(s) on which the architectural potential need to be assessed, which will be explored in phase II; and

(iii) interpreting the valuation results relative to the said objectives, which will be explored in Phase III.

Below are possible drivers for initiating the evaluation for architectural stability.

- *Valuing the cost-effectiveness of designing/re-engineering for the change.* Valuing the worthiness of reengineering or designing the architecture of the given software system to facilitate future changes in requirements,

- *Architectural risk assessment.* Risks could be due to the problematic architectural decisions. These decisions may lack the flexibility in dealing with the likely future changes in requirements. The evaluation may aim to identify the types of change(s) for which the software architecture is likely to be inflexible and likely to exhibit future threats on the stability of the architecture of the software system,

- *Software architecture trade-off.* Compare two or more candidate architectures and select the more resilient candidate to the likely critical changes in requirements.

Step I-b. Eliciting the changes $\{i_1, i_2, ..., i_n\}$

In this step, we identify likely changes, which are critical to the evaluation and to the set objectives. A question of interest is: how can we elicit or predict the change? Before we proceed in explaining the process, we define what a change is. We then identify two categories of changes: these are anticipated and extreme changes. We provide some tips from the literature for eliciting these changes.

Definition and nature of change

Change is a process that either introduces new requirements into an existing system; modifies the system if the requirements were not correctly implemented; or moves the system into a new operating environment [Yau et al., 1978; Bennett and Rajlich 2000]. Changes of requirements can be perfective, adaptive, preventive, or corrective [Bennett and Rajlich 2000]. A perfective change involves enhancing, extending, or adding/deleting the functionality of an existing system. An adaptive change requires revising requirements to properly adapt to new operating environment such as integration of a system with new hardware, peripherals, etc. A preventive change occurs when requirements are revisited to improve future maintainability, reliability, and portability or to provide a basis for future enhancements. This might include

redesigning and restructuring for requirements to rationalize system services, optimize, modularize, or create reusable components. A corrective change emerges as inadequacies, incompleteness, contradictions, ambiguities, noises, or over specification in requirements are encountered.

In software engineering, it has been known that focusing the change on program code leads to loss of structure and maintainability [Bennett and Rajlich, 2000]. Upon managing the change of requirements considerable emphasis is thus placed on the architecture of the software system as the key artifact involved [Garlan, 2000]. Managing the change is a process which involves recognizing the change through continued requirements elicitation, requirements evaluation of risk, and evaluation of systems in their operational environments [Nuseibeh and Easterbrook, 2000]. Identifying and documenting possible future changes is important in order to manage software evolution [Lehman, 1998] and evaluate architectural choices [Nuseibeh and Easterbrook, 2000]. Eliciting and dealing with the change in requirements, however, is still one of the major research challenges facing the requirements engineering community [Finkelstein and Kramer, 2000]. Some evolutionary changes could be planned. By planned evolutionary changes, we refer to changes that belong to a defined (or a semi-defined) roadmap that the system needs to accommodate in the future as part of its staged-evolution. However, other changes are unforeseen. These changes are likely to surprise the architecture as the change materializes. Below, we identify possible routes that an architect/analyst may pursue for eliciting the likely change in requirements.

Eliciting Planned Changes

Using Technology Roadmapping

Technology roadmapping is an effective technology planning tool which help identifying product needs, map them into technology alternatives, and develop project plans to ensure that the required technologies will be available when needed [Schaller, 1999]. Technology roadmapping, as a practice, emerged from industry as a practical method of planning for new technology and product requirements. According to [Schaller, 1999], a roadmap is not a prediction of future breakthroughs

in the technology, but rather an articulation of requirements to support future technical needs. A roadmap assumes a given future and provides a framework toward realizing it. Often, a roadmap is part of the business and/or the product strategy towards growth and evolution. Muller [2002] indicates that the roadmap creation process has three phases. In the first phase, a meeting is conducted to share vision on the market; and explore possible products as an answer to the market, the technology status, and the people. In the second phase, the target is obtaining a shared vision on the desired technology roadmap and analyzing a few scenarios for products, technologies, people, and process. In the third phase, a shared roadmap is created. Garcia and Bray [1997] mention an extra phase in the process, which is the follow-up activity. For this phase, all key decision makers involved are to critique, validate and accept the roadmap. An implementation plan has to be developed. This plan has to be routinely reviewed and updated. The process is a joint effort of different stakeholders, providing an opportunity for sharing information and perspectives. Stakeholders could be the business manager, the marketing manager, the technology manager, the operational manager, and the developer team including the architect(s), the requirements engineers(s), etc.

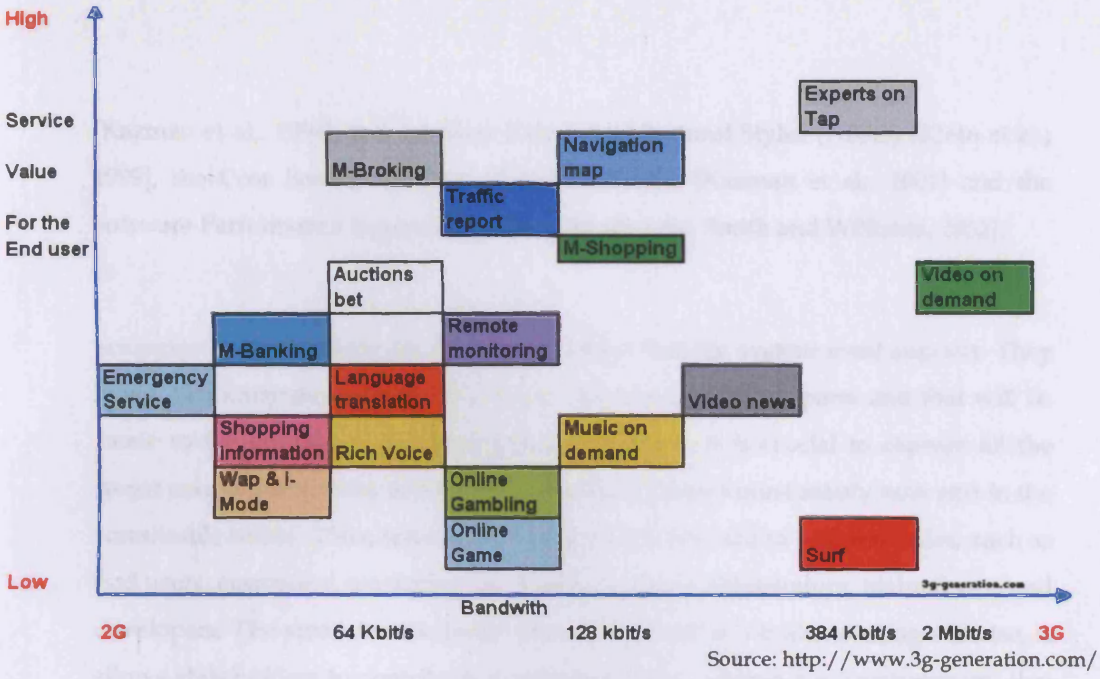
Figure 5.3 is a product roadmapping of Company x, a mobile service provider. Figure 5.3 shows how the mobile services are said to evolve as we transit from 2G to 3G networking. As the bandwidth is improved, an emerging number of content-based services, ranging from voice, multi-media, data, and location-based services might be possible. This, in turn, will translate into future requirements (functional and non-functional), which need to be planned in advance so it can be accommodated by the architecture responsible for delivering the services. Note that many of the likely changes in the requirements are often derived from the roadmapping process, rather than the roadmap itself.

As an example, M-banking is a service, which allows customers to check bank balances, view statements, and carry bank transactions using mobile phones. A distributed architecture of a banking system, which envisions providing such a service as the bandwidth is improved, may need to anticipate changes due to mobility like changes in security requirements, load, availability, etc. The architect

may then need to anticipate relevant change scenarios and ways of accommodating them on the architecture of the software system.

> E.g., M-banking availability:
> (Requirements) Loss of connectivity is the norm in mobility. The M-banking service shall be available 99% of the time,
> (Architecture) New caching mechanisms are then required.

Product-line architectures are systematic approaches for managing the change and guiding the evolution of a software system. This is achieved through anticipating the major evolutionary milestones in the development of the product, capturing the properties that remain constant through the evolution and documenting the variability points from which different family members may be created. The approach gives a structure to the product's evolution and possibly rules out some unplanned evolutions, if the architecture is respected [Jazayeri, 2000]. Product-line analysis, for example, can benefit from technology roadmapping to anticipate future requirements, and likely future product variations (which may include combinations of features not supported in current products).



Source: http://www.3g-generation.com/

Figure 5.3. Company's x technology road mapping showing the evolution of its mobile services as it moves from 2G to 3G and its value to the end user

Companies (for example, in the new communication industries) plan and envision possible paths for "perfecting" their services and offering, as the rapid advances in the technology or the infrastructure enabling these enhancements materialize. This is necessary for catching up with the market, generating wealth, and improving the value of what is offered to the end users. Moreover, these companies are investing part of their resources in envisioning the future of the stakeholders' requirements and the environment, the evolution of technology and its supporting infrastructure. This is apparent through the related investments in research and development, the increasing number of personnel recruited in technology roadmapping, and aligning the company's future performance with its ability to execute the set roadmap.

Change scenarios and change cases

The change may be exemplified using *change scenario* or *change cases*. The use of change scenarios in the analysis of software architecture has been demonstrated in a variety of evaluation methods and across a wide range of domains. In particular, change scenarios have been used in the Architecture Tradeoff Analysis Method (ATAM) [Kazman et al., 1996], the Software Architecture Analysis Method (SAAM)

[Kazman et al., 1994], the Attribute-Based Architectural Styles (ABAS) [Klein et al., 1999], the Cost Benefit Analysis Method (CBAM) [Kazman et al., 2001] and the Software Performance Engineering (SPE) [Smith 1990; Smith and Williams, 2002].

Scenarios could illustrate the kinds of activities that the system must support. They could also illustrate the kinds of changes that the client anticipates and that will be made to the system. In developing these scenarios, it is crucial to capture all the major uses of the system, and the qualities that a system must satisfy now and in the foreseeable future. Thus, scenarios represent tasks relevant to different roles, such as end users, customers, marketing specialists, system administrators, maintainers, and developers. The scenarios elicitation process by itself is a brainstorming exercise. It allows stakeholders to contribute to scenarios, in a criticism-free environment, that reflect their concerns and understanding of how the architecture will accommodate their needs. A single scenario may have implications for many stakeholders: for a modification, one stakeholder may be concerned with the difficulty of a change and its performance impact, while another may be interested in how the change will affect the integrability of the architecture.

A scenario in ArchOptions, like other architectural evaluation methods, is a brief description of some anticipated or desired use of a system. The architecture may directly support that scenario, meaning that the anticipated use requires no modifications to the architecture for the scenario to be accommodated. This would usually be determined by demonstrating how the existing architecture would behave in performing the scenario. Note that such scenarios could correspond to requirements previously addressed in the design process; hence, not "surprising" the architecture. Such scenarios may increase our understanding of the architecture, allowing systematic investigation of other architectural qualities such as performance and reliability.

ArchOptions is more concerned with scenarios that require changes to the architecture. *Growth scenarios* represent ways in which the architecture is expected to accommodate growth and change in the moderate near term. These may include

expected modifications, changes in performance or availability, porting to other platforms, integration with other software, and so forth. Growth scenarios provide a way to show the strength and the weakness of the architecture with respect to anticipated changes.

If the scenario requires modification to the architecture; these changes could be related to how one or more components perform an assigned activity; the addition of a component to perform some activity; the addition of a relation between existing components; the removal of a component or a relation; a change to an interface; or a combination of these. These types of scenarios are often referred to as *indirect scenarios.* An indirect scenario is a one that requires a modification to the architecture to be satisfied. Indirect scenarios are central to the measurement of the degree to which an architecture can accommodate evolutionary changes. The cumulative impact of indirect scenarios on an architecture measures its suitability for ongoing use throughout the lifetime of the family of related systems. Directed scenarios are similar to use cases in UML notation and indirect scenarios are sometimes known as change cases.

Note *Use cases* of the Unified Modeling Language (UML) may provide an alternative for representing the change. For example, we may build on use cases to integrate both time and "variability" information. The overall outcome may "visualize" the change and facilitate communicating it to the concerned parties.

Dealing with the extreme changes

If changes can be predicted, then they can be anticipated in the design. The hard problem, thus, is coping with *extreme* changes. As for this category of changes, we acknowledge the fact that there are no silver bullets for precisely and efficiently eliciting these changes, their variation over the lifetime of the software system, and their likelihood. We rely on *exploratory scenarios* [Kazman et al., 1996] for predicting classes of possible changes. *Exploratory scenarios* exemplify "dramatic" changes, which *if they occur,* may stress and surprise the architecture of the software system. These changes may take the form of extreme growth that are likely to "stress" the

system, such as dramatic changes in scalability, performance, availability requirements, and major changes in non-functional requirements.

Exploratory scenarios attempt to find sensitivity points that appear to stress the architecture. The identification of these points helps assess the limits of the architecture, hence optimizing the chances of surfacing the architectural decisions to risks.

Step I-c. Trace the change to the architecture

This step constitutes of the following activities:

Identify goals from scenarios

E.g., Use heuristics and guidelines suggested by [Anton, 1997] to identify the goals

Trace the goals to the architecture

Refine the goal using knowledge of the solution domain until a trace with the associated architectural artifacts, which implement or said to be impacted by the change, is established.

The output of the previous step is likely change(s) that need to be accommodated or could surprise the architecture. The changes are said to be exemplified using scenarios. In this step, we want to understand how the changes relate, are realized, or could impact the architecture of the software system. The objective is to quantify the cost of the change and value the architectural flexibility relative to the change, which we will explore in Phase II.

A scenario could hold a rich description of the likely change(s) to the software system. A brief scenario, however, could be further refined to correspond to one or more further changes that may need to be realized or could impact the architecture of the software system. Note that ArchOptions is more concerned about *how* the *goals* of a given scenario are "operationalized" or could affect the architecture of the software system. The objectives are (i) to provide a paradigm, which traces the change in the requirement, exemplified by the scenario, into its architectural

elements, and (ii) to quantify the flexibility of the software system in responding to the scenario exemplifying the change. Though existing architectural evaluation make use of scenarios, they lack the support for systematically analyzing and approximately tracking scenarios into the architecture of the software system. In existing architectural evaluation methods, the architect explains how relevant architectural decisions contribute to realizing a particular scenario. Ideally, this activity is dominated by the architect in explaining how the architecture generally addresses a particular scenario.

One possible strategy for tracing the change in requirements to the architecture of the software system is to build on Goal-Oriented approaches to Requirements Engineering (GORE) [e.g., van Lamsweerde, 2000]. According to [van Lamsweerde, 2000], *goals* are prescriptive statements of intent whose satisfaction requires the cooperation of *agents* (or active components) in the software and its environment. Goals may be organized in structures that capture how they are being refined or abstracted. Such structures form the skeleton of goal models: goals there range from high-level, strategic objectives to fine-grained, technical prescriptions that can be assigned as responsibilities of single agents. Goals may refer to *functional concerns* or *quality attributes*. A functional goal typically captures some maximal set of desired scenarios. A quality goal typically captures some preferred behaviors among those captured by functional goals. An appreciated feature of GORE models is their built-in vertical traceability - from strategic business objectives to technical requirements to precise specifications to architectural design choices. The ability to capture multiple system versions within the same model through multiple paths of the goal graph (e.g., the system as-is, to-be, and likely-to-be-next) are helpful in case of tracing the high-level goals into the corresponding architectural elements.

Briefly, our use of the goal-oriented approach is general. We adopt a goal-oriented approach to refine the requirements (e.g., [Dardenne et al., 1993; Anton, 1996]). We derive goals from scenarios (e.g., using some heuristics suggested in [Anton, 1997]). We then refine the goals using knowledge of the solution domain until a trace with the associated architectural artifacts, which implement or are said to be impacted by the change, is established. The process is fairly simple and involves following two major steps:

Identifying goals from scenarios

We analyze the scenarios to identify goals that need to be met by the software system's architecture. Goal analysis began by identifying goals in the scenarios [Anton, 1997]. Anton [1997] provides a methodology and heuristics for identifying goals from scenarios. Representative examples can be found in Table 5.1.

Table 5.1. Some useful heuristics for identifying goals from scenarios - Anton [1997]

No.	Heuristic
H1	Key action words such as: track, monitor, provide, supply, find out, know, avoid, ensure, keep, satisfy, complete, allocate, increase, speedup, improve, make, and achieve are useful for pointing to candidate goals
H2	Action words that point to some state that is or can be achieved once the action is completed as candidates for goals. They are identified by considering each statement in the scenario by asking: Does this behavior or action denote a state that has been achieved, or a desired state to be achieved? If the answer is yes, then express the answer to these questions as goals, which represent a state that is desired or achieved within the system
H3	An effective way to uncover hidden goals is to consider each action word and every description of behavior and persistent ask "why?" until all the goal have been "treated" and the analyst is confident that the rationale for each action is understood and expressed as a goal. The action words should be restated so that they denote a state that has been achieved or a desired state.
H4	If a statement seems to guide decisions at various levels within the system or organization, express it as a goal
H5	Stakeholders tend to express their requirements in terms of operations and actions rather than goals. Thus, when given an interview transcript, it is beneficial to trace action word strategy to extract goals from stakeholders' descriptions
H6	Customers tend to express their goals within the context of their application domain, not in terms of an existing or desired system. Analysts should first seek to understand the stakeholders' application domain and goals before concentrating on the actual or the current system so that the system requirements may be adequately specified.

We shall not go into much detail, as the process is intuitive and outside the scope of the thesis.

Trace the goals to the architecture

In this step, we refine the goals and identify the sub-goals. In the refinement process, the goals are decomposed into more concrete subgoals, which correspond to richer and more tangible representation of the parent goals. In ArchOptions the refinement is done using guidance on how it could be operationalized by the architecture. In

more abstract terms, the guidance is given by the knowledge of the domain, vendor's specification, related design and implementation experience, related design patterns, etc., and in association with the solution domain (e.g., the underlying middleware). Another objective of the refinement process is to make goals corresponding to the change as measurable as possible to quantify the costs and benefits associated with the change. The refinement of goals continues until they can relate the change to architectural components, strategies/mechanism (i.e., the architectural element that said to operationalize the change) and until we will be able to measure its corresponding impact on the architecture of the software system.

The refinement process could result in: (i) identifying the architectural elements responsible for operationalizing the goals; or (ii) identifying architectural elements which might be impacted by the change. Note, treating goals which represent changes in a functional nature is obviously less demanding than goals of non-functional nature, as the change is often localized in a set of architectural elements. Goals of a non-functional nature are more critical as they can have a global impact on the architecture.

The goal refinement graph could capture relationships among goals using and AND/OR refinement links. AND refinement relates to goals that are satisfied when all its subgaols are satisfied. OR refinement relates to a goal which is sufficiently satisfied if at least one of its subgoals are satisfied. Note that different architectural mechanisms may operationalize a given goal, which may be captured in the AND/OR graph or by a general graph.

Back to our running example, obviously the goal that could be extracted from the scenario narration is *maintaining scalability*. Figure 5.3 shows the goal-oriented graph refinement corresponding to the change in scalability. In Chapter 6, we will see that the refinement was guided by the knowledge of the domain (i.e., the middleware primitives); vendor's specification, such as [Object Management Group, 1999-2000; Sun Microsystems Inc., 2002]; related design and implementation experience, mainly that of [Othman et al., 2001a; Othman et al., 2001b]. The scalability goal was refined into two major sub-goals: these are achieving load-balancing and fault tolerance on

the architecture of the software system. Note that different architectural mechanisms may operationalize the scalability goal and its corresponding refinements. As an operationalization choice, we use replication as way for achieving scalability. The reason is due to the fact that both CORBA and J2EE provide the primitives or guidelines for scaling a software system using replication. We have relaxed the use of AND/OR representation as we are modeling the system as-is with one operationalization choice.

Consider the Fault Tolerance sub-goal of Figure 5.3: the requirements for implementing Fault Tolerance and their CORBA architectural realization are depicted in Table 5.2. They are refined based on the CORBA fault tolerance specification of the OMG [Object Management Group, 1999]. Detailing the refinement and the operationalization of the goal can be found in Chapter 6 with the complete case study.

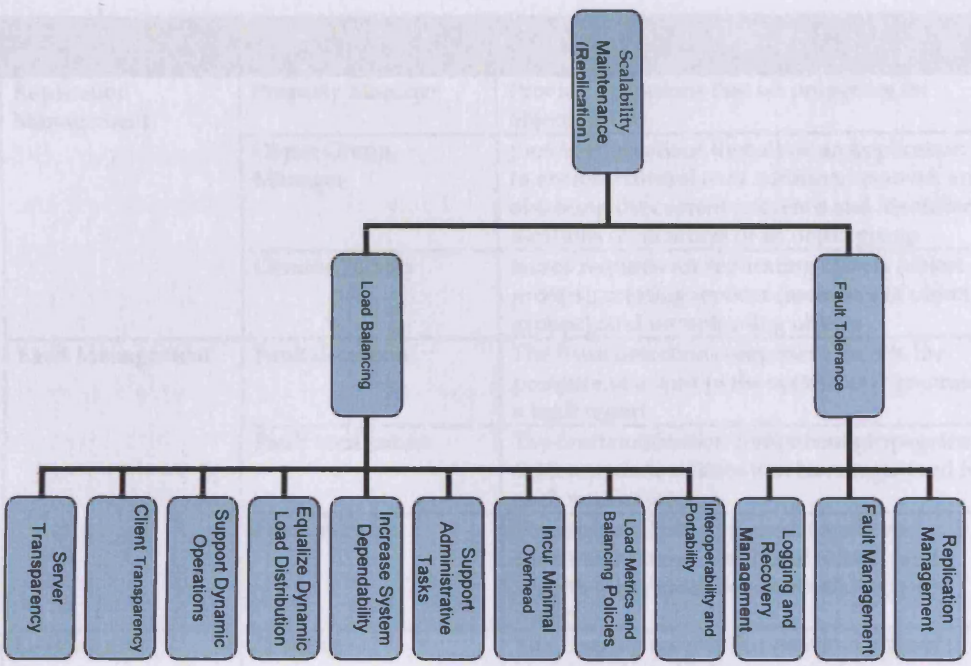


Figure 5.3. The goal-oriented refinement for achieving scalability through replication

Table 5.2. The refinement of the fault tolerance subgoal (CORBA)

Sub goals	Architectural Elements	Description
Replication Management	Property Manager	Provide operations that set properties for object groups
	Object Group Manager	provide operations that allow an application to exercise control over addition, removal, and obtaining the current reference and identifier locations of members of an object group
	Generic Factory	Issues requests for replicating objects (object groups), creating replicas (members of object groups), and unreplicating objects
Fault Management	Fault detection	The Fault detection component detects the presence of a fault in the system and generates a fault report
	Fault notification	The fault notification component propagates fault reports to entities that have registered for such notifications
	Fault analysis	The fault analysis component analyses a (potentially large) number of related fault reports to generate a condensed diagnosed report
Logging and Recovery Management	Logging	The Logging records the state and actions of a member of an object group in a log
	Recovery	The Recovery Mechanism sets the state of a member, either after a fault when a backup member of an object group is promoted to the primary member, or alternatively when a new member is introduced into an object group

5.2 Phase II. Valuing the Flexibility of the Architecture to the Change

The problem of valuing the flexibility of an architecture to likely changes in requirements needs a comprehensive solution that is flexible enough to incorporate multiple valuation techniques; some with subjective estimates and others based on market data, when available. This is because of the following reasons:

First, the valuation activity is a *human-centered activity*. The participants in the valuation activity may include developers, architects, project mangers, market analysts, product analysts etc. Interviews, meetings, or surveys could be conducted to gather qualitative and quantitative costs and benefits information. The participants often rely on experience and subjective judgments in valuation. Describing the valuation as human-centered activity implies subjectivity and introduces different perspectives to the valuation problem.

Second, the change may impact one or more architectural qualities, such as performance, maintainability, availability and so forth when need to be accommodated by the system of a given architecture. For example, Chapter 6 demonstrates a case where a change in scalability requirements affects both behavioral and structural qualities of an architecture. Linking the impact of the change to value, as a way for valuing flexibility, requires a valuation solution that is comprehensive enough to account for the economic ramifications of the change and its global impact on the architecture including how the change could affect one or more architectural qualities. The aim is to provide the architect/analyst with a comprehensive tool for understanding the extent to which the change can "ripple" to impact other qualities and its economic implications.

Third, technically speaking, real options valuation uses twin asset to the valuation of the asset in question. If the twin asset is not directly observable, it is reasonable to use estimates of return on the asset in question to estimate value or market-calibrated value [Schwartz and Trigeorgis, 2000]. In some cases, the flexibility of the architecture to change in requirements can be valued in terms of directly observable

cash flows linked to future operational benefits or market value, making it easy to use the return to value the options. In other cases, the flexibility of an architecture to the change may not be directly observable through cash flows. Consequently, the analyst may then need to rely on experience for estimation. If the analyst relies on experience and judgment in his/her estimation, the estimates tend to be subjective but could make an implicit use of market information. Note that back-of-the-envelope calculations, which are based on value estimates (rather than on market value), continue to be acceptable and revealing [Sullivan et al., 2001]. It is often the case that both market and subjective value estimates are available. That is, in real options, values are often estimated by inspecting a relevant experience or by using subjective estimates. Hence, this brings a need for a solution that comprises both value and accounts to the different perspectives to the valuation.

Fourth, the valuation is relative to the evaluation objectives, set in Phase I and the primary drivers motivating the change. The drivers could be, for example, future cost savings, shorter time-to-market, entry to new markets, service enhancements, etc. It is often the case that there is more than one driver behind the change. This necessitates a valuation solution that is flexible enough to capture the value relative to the said drivers.

As a compromise, the problem of valuing the flexibility of an architecture to a likely change necessarily requires a comprehensive solution that is flexible enough to capture the options from different perspectives and to incorporate multiple valuation techniques; some with subjective estimates and others based on market data, when available. The problem of how to guide valuation and introduce discipline in this setting, we term as the *multiple perspectives valuation problem*. To address this problem, we outline a conceptual *valuation points of view* framework. The framework aims to capture and value the flexibility of the architecture to change from different *points of views*. A *point of view, P,* is a perspective used by an analyst/architect to assess the architectural potential to the change. The perspective could be either technically related (e.g., structural such as development, configuration, deployment; behavioral such as performance, availability, reliability etc.), market-related (e.g., market potential of a product), and/or related to the organization business objectives. Therefore, the corresponding value of an architectural potential to a change may be

relative to the market, to one or more technical dimension of the system, or to the organization, as sketched in Figure 5.4. The purpose is to reach a comprehensive value of options from different perspectives. In addition, the aim is to promote flexibility through incorporating both subjective estimates, which may implicitly use market information and/or explicit market value, when available. Furthermore, it remains an open challenge to strongly justify precise estimates for real options in software [Sullivan et al., 2000]. Part of the problem stems in the absence of frameworks that capture the options on the software from different perspectives. The outlined valuation point of view framework is promising to address these shortcomings.

Steps II-b develops on how we can value an architectural potential to change relative to a point of view. We define and discuss two valuation points of view: these are technical and market valuation points of view.

For a valuation point of view p_j and a change i, the constructed call options could be re-expressed in *(5.1)*, where $x_i Vp_j$ corresponds to the value of the architectural potential of the change relative to p_j, with an exercise cost of $C_{ei}p_j$:.

$$E\,[\max\,(x_i Vp_j - C_{ei}p_j,\, 0)] \qquad (5.1)$$

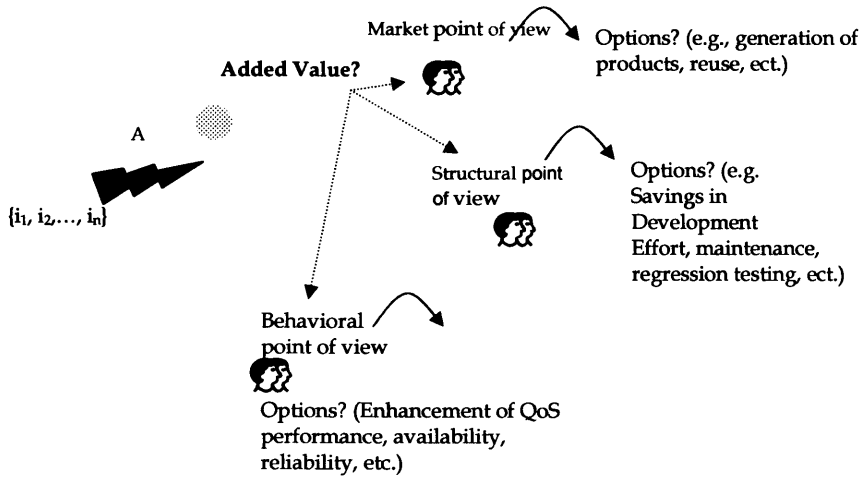


Figure 5.4. Valuing the options using valuation points of view for changes $\{i_1, i_2,\ldots, i_n\}$ on architecture A

In context of architectural stability, a potentially stable architecture has to maximize the value added relative to some valuation points of view. In Chapter 6, we will see how the decision of selecting an architecture which tends to be more accommodating for changes in scalability requirements has taken into account both the value added relative to two valuation points of views. These are maintainability (structural) and throughput (behavioral) (Section 6.3).

Phase II constitutes the heart of the ArchOptions model. In this phase, we identify the valuation points of view on which the options will be computed. For a valuation point of view p_j: we analyze and list the changes that are necessary to be performed on the architecture. We estimate the cost of accommodating the change. This cost corresponds to the exercise price. We value the potential of the architecture to withstand the change. We analyze ways for computing the fluctuation in the estimated value. At the end of Phase II, the major inputs of the ArchOptions model would have been identified. These are $x_i V_{pj}$ (i.e., Value of the "architectural potential" in supporting the change), σ_{pj} (i.e., the "fluctuation" in the return of value of $x_i Vpj$),

and C_{epj} (i.e., the estimate of the likely cost to accommodate the change) and relative to the valuation point of view *pj*. Having these parameters, we can then construct calls to value the flexibility of the architecture to the change.

The steps below constitute phase II, which we detail in the following subsections.

Step II-a. Estimate C_{eipj}

Estimate the cost of the architectural strategy, mechanisms, and/or the associated implementations, which realize the change- the cost corresponds to the exercise price

Let us return to our running example: The cost of realizing scalability with the CORBA-induced architecture, translates to the cost of building a replication mechanism, responsible for realizing the changes in the scalability goal. In concrete terms, the cost materializes to the cost of implementing load balancing and fault tolerance services, configuration of these services, and deployment of the replicas running these services on hosts. These may translate into development cost (i.e., person-months), hardware, licensing costs (if any), etc. In abstract terms, the change materializes to an architectural strategy or mechanism responsible for realizing the said goal. Moreover, the change may affect the existing architectural components, connectors, and/or the underlying infrastructure requiring modification to the associated software artifacts. Generally speaking, ArchOptions is flexible to incorporate either coarse-grained or fine-grained cost estimation. Note that the ArchOptions model is complementary to expert estimation, where expert estimates of the change can be fed into our model. To help experts come up with estimates that are more precise, they can inspect relevant effort, past projects, associated design patterns, and so forth. Alternatively, techniques such as COCOMO II [Boehm et al., 1995] may be used if the key predictors, such as size of the change can be reliably estimated. As with expert-based estimation, the estimates for change could be fed into the model. Note that by inspecting a previous valuation experience to satisfy the concept of "twin asset" and by identifying the key predictors to COCOMO II, we end up applying a "composite" approach to cost estimation. An approach which combines both expert knowledge and parametric estimation is said to be more precise than approaches which solely rely on either expert knowledge or parametric models to estimation [Briand and Wieczorek, 2002].

For example, Table 5.3 shows how the *Fault Tolerance* subgoal refinement relates to the JAVA classes implementing the change. Table 5.3 estimates that SLOC required to implement the change using an analogy with a previous development experience. Using the SLOC, we can then estimate the cost using models like COCOMO II [Boehm et al., 1995]. However, the real-world usefulness of models such as COCOMO II has been questioned for constant and unexplained calibration, which often leads to inaccuracy in the prediction. It could be also argued that in iterative development, when estimations are continuously recalibrated (e.g., in the Unified Process), it is possible to come up with estimations that are more accurate than COCOMO II, as they will take into account factors, such as the skills of the developers, the project maturity, and other organizational factors.

Table 5.3. Implementing the fault tolerance service on CORBA

CosFaultTolerance	IDL	242	Interface description of remote methods
PropertyManagerImpl	Java	273	Implementation of the PropertyManager interface
ObjectGroupManagerImpl	Java	672	Implementation of the ObjectGroupManager interface
GenericFactoryImpl	Java	523	Implementation of the GenericFactory interface
ReplicationManagerImpl	Java	865	Implementation of the ReplicationManager interface
FaultNotifier	Java	611	Implementation of the FaultNotifier interface
ClientPolicy	Java	155	Implementations of the RequestDurationPolicy interface
ServerPolicy	Java	61	Implementation of the HeartbeatEnabledPolicy
FTPolicy	Java	207	Implementation of the HeartbeatPolicy interface
FaultDetector	Java	149	Class defining the component illustrated above
DefaultFaultAnalyzer	Java	113	The default fault analyzer
ReplicationManagerFaultAnalyzer	Java	865	Replication Manager's fault analyzer
FaultConsumer	Java	200	Connect to the fault notifier
PropertyValidator	Java	29	Class providing static methods to validate properties
MemberInfo	Java	50	Structure that contains all member-specific information
PropertyUtils	Java	53	Provides some methods used to manipulate properties
Operators	Java	23	Class providing static methods related to operators
ReplicationManagerServer	Java	13	Class running the Replication Manager server
FaultNotifierServer	Java	13	Class running the Fault Notifier server
Total		5117	

Generally speaking, for estimating the exercise cost, three possible routes can be pursued:

(i) Use expert knowledge to cost estimation, or

(ii) use parametric models to cost estimation, or

(iii) combine expert knowledge with parametric models for better estimation. Note that in [Briand and Wieczorek, 2002], the prediction accuracy of several cost estimation models has been reviewed. Examples include the Constructive Cost Model COCOMO [Boehm, 1980], Ordinary Least Squares (OLS) regression [Subramanian and Breslawski, 1993], and ANALOGY [Walkerden and Jeffery, 1999]. The review examines the

results of several empirical studies done in the last fifteen years to evaluate the prediction effectiveness of the subject models. The result shows that the estimation using these models could be improved if their parameters are adjusted using expert knowledge.

Expert knowledge to cost estimation

Expert knowledge, also referred to as non-model based estimation methods, consists of one or more estimation techniques together with a specification on how to apply them in a certain context. These methods do not involve models but rely on direct estimation. Obviously, they require heavy involvements of experts, their previous experience, and judgment to generate an estimate of the cost for implementing the change. Using solely non-model based methods may lead to very inaccurate results. Developers may tend to underestimate the time required to do small changes, yet they tend to overestimate the time for larger ones [Briand and Wieczorek, 2002]. Expert based techniques are typically best suited for projects that are not too different from the projects completed in the past. The analyst may have developed an extensive experience in similar situations, which makes it easier to estimate. The main drawback, however, is the subjective and the non-transparent nature of the estimation process that make it harder to justify the estimates. Often it is difficult to find analysts with the appropriate experience in the application and the environment in which the change needs to be developed.

Parametric models to cost estimation

Software development costs continue to increase and practitioners continually express their concerns over their inability to accurately predict the costs involved. As a result, the software engineering community has been concerned with the development of models that constructively explain the development life-cycle and predict the cost of developing a software product since the early 1960s. The field of software engineering cost models, however, has had its own pitfalls: the fast changing nature of software development has made it very difficult to develop parametric models that yield high accuracy for software development in all domains. Model-based or

parametric-based estimation is usually dependent on a number of inputs (e.g., a size estimate, cost factors) and outputs an effort point estimate or distribution.

Throughout the thesis, we use COCOMO II [Boehm et al., 1995], as a parametric model to estimate cost. Appendix A provides the interested reader with a quick overview on COCOMO II.

Step II-b. Estimate $X_i V_{pj}$

Using the valuation objectives, identify the value of the architectural potential with respect to the change

Upon the application of the model, the problem that the analyst/architect faces is that the cost is often tangible, but the value is hard to grasp. For example, refactoring a system of a given architecture incurs up-front design costs; but the value is so elusive and long-term. Part of the value may materialize if the refactoring exercise is planned so the structure can be utilized to create future value such as future savings in maintenance and regression testing. Such a value may span several dimensions such as ease of future maintainability, extensibility, modularity, reusability, complexity, and efficiency. Returning to our running example we have highlighted in Chapter 4, the value of the architectural potential of inducing an architecture with J2EE and not CORBA (and vice versa) is a relative value. The value could span different dimensions including ease of future maintenance and relative savings in deployment and configuration of the software system if we choose to go for J2EE and not a CORBA-induced architecture (and vice versa). This value is realized only if the change in future load materializes. Alternatively, the architectural potential could be valued in relation to the market, as it is the case with product line-architectures. For example, the architecture could "pull" the options by responding to changes in the market requirements, while leaving the architecture of the software system intact or by requiring minimal changes to the architecture. In many cases, the value crosscuts many dimensions ranging from market to technical leading to both technical and market benefits.

Hence, the valuation is relative to the evaluation objectives, set in Phase I and the primary business drivers motivating the change. The business driver could be for example, future cost savings, shorter time-to-market, entry to new markets, service enhancements, and so forth. In many cases, we consider that the right to claim future cost savings as a result of the architecture supporting the change is a value. In other cases, the value of the architectural potential is a consequence of an upfront investment to facilitate future changes, which in turn will create value. The payoff occurs in the future, contingent on uncertain future conditions. It is worth noting that valuing the architectural potential is case dependent and there is no generic off-the-shelf solution to such valuation. The valuation activity is a human-centered activity. Ideally, the valuation is done in connection with the product, strategy, and/or the marketing team.

We discuss how we can value an architectural potential to change relative to a point of view. We discuss two valuation points of view: these are technical and market valuation points of view.

Valuation using technical point of view

By using a *technical point of view* to assess the architectural potential to the change, we may aim at assessing the architectural potential of an architecture to the change relative to some structural or behavioural properties of the system of a given architecture. As an example of the structural properties, we may aim at assessing the expected savings (if-any) in development, configuration, and deployment efforts to be realised upon accommodating the change on the system of a given architecture. We may also be interested in assessing savings in licenses and hardware. For the behavioural properties, we may for example, aim at understanding the economics implication of the change on one or more architectural qualities such as performance, reliability, availability, and so forth. Chapter 6 provides an extensive example on how both structural and behavioural valuation points of view are used. In many other cases, the enterprise could focus the analysis on one technical dimension. For example, by using *development point of view* to assess the architectural potential to the change, we may aim at understanding the savings in development effort (if any) to be realised upon accommodating the change on the system of a given architecture.

Therefore, the value of the architectural potential to the change could be realized in relation to one or more technical dimension. In fact, the choice of the dimensions is dependent on how the enterprise defines its value proposition. As a result, there is no generic off-the-shelf formula. A range of metrics can be used. Typical measures may include cost savings; risk and losses avoidance; increased productivity; reduction in personnel required for integration; reduction in time-to-market; savings in regression testing effort; and/or enumeration of short-term (e.g., quarterly cycle) and long-term (e.g., two-years or more) benefits and so forth. Our assumption here is that the resulting value is cast into monetary value.

Valuing the architectural potential to the change requires finding a twin asset with the similar risk characteristic of the one at hand. We have argued that reusing a past development experience such as previous design and its corresponding implementation to inform the valuation bear a resemblance to the concept of a "twin asset" [Bahsoon et al., 2005; Bahsoon and Emmerich, 2004a; Bahsoon and Emmerich 2004b]. Note that much of the valuation in software engineering is based effort measured in person-months. Such valuation is based on similar experience and may hold similar risk characteristics to the case in hand. The valuation does implicitly hold market information as effort valuation if often priced relative to the market. Back to our motivating example, as we will see in Chapter 6, that in valuing the architectural potential of the CORBA-induced version relative to that of J2EE, we have used a previous design and development experience, where the scalability change has been designed and implemented on a CORBA compliant middleware, TAO (refer to Chapter 6). In this context, our use for the design and the corresponding implementation of scalability on TAO bears a resemblance to the concept of a "twin asset", for we are reusing a past development experience to inform the valuation. To value the x_iV of the J2EE induced-architecture, S_1, relative to the CORBA induced-architecture, S_0, in responding to the change in load, we take a *technical point of view* to valuation. The valuation uses the expected savings (if-any) in development, configuration, and deployment efforts, when the change in load needs to be accommodated on S_1 relative to S_0, and respectively denoted as $\Delta_{S1/S0}C_{dev}$, $\Delta_{S1/S0}C_{config}$, $\Delta_{S1/S0}C_{deploy}$. Relative savings in licenses and hardware may also be considered and respectively denoted by ΔC_{licesh}, ΔC_{hardw}. Below is a model for

calculating $x_i V_{S1/S0}$ relative to the change, expressed in cumulative savings for h hosts:

$$x_i V_{S1/S0}\ technical\ point\ of\ view = \sum_{h=1...k} (\Delta_{S1/S0} C_{dev}, \Delta_{S1/S0} C_{config}, \Delta_{S1/S0} C_{deploy}, \Delta_{S1/S0} C_{licesh}, \Delta_{S1/S0} C_{hardw})_h$$

Alternatively, the analyst/architect may break down the valuation relative to one point of view at a time. Table 6.11a of Chapter 6 provides an example on using the technical point of view and breakdowns of the calculations relative to a particular point of view as expressed below:

$$x_i V_{S1/S0}\ Development\ point\ of\ view = \sum_{h=1...k} (\Delta_{S1/S0} C_{dev})_h;$$
$$x_i V_{S1/S0}\ Configuration\ point\ of\ view = \sum_{h=1...k} (\Delta_{S1/S0} C_{config})_h;$$
$$x_i V_{S1/S0}\ Deployment\ point\ of\ view = \sum_{h=1...k} (\Delta_{S1/S0} C_{deploy})_h.$$

In the refactoring case of Chapter 6, we restrict the valuation to one point of view, the *development point of view*. The objective is to value the improved architectural potential as a result of investing in a refactoring exercise. The architectural potential was assessed relative to likely savings if twenty changes, *ch*, of adaptive nature may need to be accommodated on the refactored version.

$$x_i V_{S1/S0}\ Deployment\ point\ of\ view = \sum_{ch=1...20} (\Delta_{S1/S0} C_{dev})_{ch}$$

Valuation using the market point of view

The value of the architectural potential could be realized in relation to the market or the enterprise business objectives. This is true when the change is driven by purely market needs: this could be in response to market differentiators, assimilating and exploiting new technologies, in response to changes in standards, customer

demands, and market competition. By using a *market point of view* to valuation, we may aim at assessing the market potential of the architecture upon supporting the change leading to new products, new services, etc. The market point of view may provide an insight on the profitability of evolution and consequently the success (failure) of evolution relative to the market upon accommodating the change.

The analysis may highlight the role of the architectural flexibility in instantiating from the core architecture new market products. This gives the analyst/architect a way to think about this flexibility as being tangible. The analysis may provide an answer to when the payback will be realized upon investing in the change.

We have exemplified the use of the market valuation point of view to value the flexibility of a small product-line suite, xlinkit [www.systemwire.com], in responding to changes in the market requirements. The change is driven by a need to accommodate a new market standard. In summary, the xlinkit suite provides capabilities for checking the consistency of distributed and heterogeneous documents. xlinkit uses a built-in grammar-based Extensible Markup Language (XML) validation language, referred to as CliX, to the consistency checking and the validation of these documents. Being a grammar-based validation language, CliX has some limitations when validating complex documents, which are inconvenient and difficult to represent using grammar-based languages. Example of this category of documents is patterns of graph-structured data of scholarly research. Schematron [Jelliffe, 2000; Miloslav, 2000] is a unique grammar-free validation language that is suitable for validating this category of documents. The current xlinkit implementation does not support Schematron. As Schematron is undergoing ISO certification, Schematron is likely to become one of the most used XML validation languages in the market. For xlinkit, the support of Schematron is likely to enhance the product potentials for the capability of CLiX and Schematron are complementary. This is in turn may translate into long-term revenues for the enterprise due to likely penetration of new markets. We have shown how ArchOptions can value the flexibility of the core xlinkit architecture in integrating Schematron. The objective of the case is to exemplify the use of the valuation points of views framework. Upon valuation, we have appealed to the use of two valuation points of views: the maintenance and market valuation points of views. The analysis has shown a

possible way of using ArchOptions to provide insights into the likely success (failure) of the software evolution and its implication on the software system. The case has provided an idea on how ArchOptions can be employed to quantify the value of the architectural potential in supporting new market product while achieving a net benefit. The interested reader may refer to [Bahsoon and Emmerich, 2005] for more details.

Using the *market point of view* to value the architectural potential has some shortcomings

> *Limited applicability.* The only time where an architectural potential can be assigned a market value is when the resulting product due to introducing new feature can be sold, create market revenues, or be correlated with the market.
>
> *The valuation is subject to manipulation and fairly subjective.* This is because the valuation could be affected by variation in the market conditions such as supply and demand, market competition, contractual agreements etc. This often leads to subjectivity upon assigning a market value.

A question of interest, however, how could we capture such value? In real options, values are often estimated by inspecting a previous relevant experience or by using subjective estimates. The participant in the valuation activities may include the developers, the architects, the project mangers, the market analysts, and other stakeholders. Interviews, meetings, or surveys are often conducted to gather benefit information. It is the norm that enterprises construct business cases for justifying the upfront investment in a particular architecture. In some cases, a business case may include some probable evolutionary milestones in the lifetime of the architecture, forecast of possible revenues, enumeration of some benefits, risks, and so forth. The business case may also include estimates of costs and valuation scenarios for probable payback upon realizing the evolutionary milestones, such as instantiating from the core architecture a new market product. If this is the case, the use of valuation scenarios to capture the possible value of the architectural potential upon accommodating the change over a period of interest becomes feasible. The scenario

valuation preserves the dynamisms entailed by the options approach and accounts for various possible and foreseen values.

Figure 5.4, For example, depicts an extract from Company Y's valuation of the probable payback upon instantiating from the core architecture a simplified new market product and in response to market requirements. The valuation uses five scenarios showing a likely payback value ranging from £-15,028(Scenario 3), £14,025(Scenario 1), £37,472(Scenario2), £40,472(Scenario 4), to £55,153(Scenario 5). Note that these values correspond to the present value:

$$x_i V_{\text{market point of view (scenario 1)}} = £14{,}025;$$

$$x_i V_{\text{market point of view (scenario2)}} = £37{,}472;$$

$$x_i V_{\text{market point of view (scenario 3)}} = £\text{-}15{,}028;$$

$$x_i V_{\text{market point of view (scenario4)}} = £40{,}472;$$

$$x_i V_{\text{market point of view (scenario 5)}} = £55{,}153.$$

Business Case for PAML

Income

	FTB	Amex Bank	EFG	PAML1	Co-op
Licence	$250,000	$243,000	$447,100	$250,000	£360,000
PS charge to customer	$100,000	$50,000	$300,900	$100,000	£460,000
PS income after costs (cost = $59,500 assuming 100 man day deployment)	$41,500	-$9,500	$241,400	$41,500	£400,500
Maintenance	$54,000	$52,488	$96,574	$54,000	£77,760

% Income		FTB	Amex	EFG	PAML 1	Co-op
Product Marketing Licence	10.2%	$25,500	$24,786	$45,604	$25,500	£36,720
PS	35.2%	$14,606	-$3,344	$84,973	$14,606	£140,976
Corp-License	36.0%	$90,000	$87,480	$160,956	$90,000	£129,600
Corp-Services	9.5%	$3,943	-$803	$22,933	$3,943	£38,048
Corp-Maintenance	8.5%	$4,590	$4,461	$8,209	$4,590	£6,610
TOTAL INCOMES		**$138,641**	**$112,481**	**$322,676**	**$138,641**	
TOTAL INCOMES£		**£81,553**	**£66,165**	**£189,809**	**£81,553**	**£361,953**

Expenditure

Goal total Man Days	848
Daily Internal Charge Rate	£350
Total Expenditure	**£296,800**

Income

Payback Scenario 1			
FTB+AMEX	£147,719		
PAML1	£81,553		
PAML2	£81,553		
Total income	**£310,826**	**Profit =**	**£14,026**
Payback Scenario 2			
FTB+AMEX	£147,719		
PAML1	£81,553		
2 AMLE deals	£105,000		
Total income	**£334,272**	**Profit =**	**£37,472**
Payback Scenario 3			
FTB+AMEX	£147,719		
PAML1	£81,553		
1 AMLE deal	£52,500		
Total income	**£281,772**	**Profit =**	**-£15,028**
Payback Scenario 4			
FTB+AMEX	£147,719		
EFG	£189,809		
Total income	**£337,527**	**Profit =**	**£40,727**
Payback Scenario 5			
Co-op	£351,953		
Total income	**£361,963**	**Profit =**	**£56,163**

Figure 5.4. An extract from Company Y's valuation of the probable payback upon instantiating from the core architecture a simplified new market product

Another simplified solution would be using value estimates representing *pessimistic*, *optimistic*, and *likely* [Gilb, 1998] values of the architectural potential, over a specified period of interest. We demonstrate the use of such a solution in the evaluation section of the thesis.

Calculate σ_{pj}*:*

> *E.g.*, Estimate the likely variation in the optimistic, likely, and pessimistic value,
> Alternatively, Estimate the likely variation in valuation scenarios,
> Compute the standard deviation of the elicited variations
> Alternatively make a modeling assumption of σ OR make an educated guess of σ

In short, the volatility σ_{pj} tends to provide a measure of how the stakeholders are uncertain about the future value of the architectural potential relative to the change and relative to p_j; it tends to measure a fluctuation in value. In financial options, practitioners often rely on historical data of investment returns to estimate the volatility of the stock price. This is feasible because the valuation is done in span of the market where high volume of historical data is available. Yet, this is not the case in valuing software. For example, the case of valuing the architectural potential to the change may hint that the uncertainty and the fluctuation in value are private to the given project. Further, such case often occur in low volumes, therefore getting valid data, treating them consistently, and dealing with the non-quantifiable effects makes the valuation and estimating volatility different from market-traded options. Hence, unlike financial options where richly traded-market information on values and uncertainty are available, it is hard to provide reliable and justified estimates of volatility in real options. Note that real options practitioners often rely on subjective opinion to estimate the volatility. In many cases, real options practitioners make simplified assumptions by either using modeling assumptions or making educated guess. For example, one approach is to examine a range of estimates from say 30% to 60% and guess which might be the most appropriate. When the estimates are poorly justified, performing sensitivity analysis to verify the choice becomes essential.

In modeling volatility, in some cases we adopt a simplistic solution to the problem. We use stakeholder judgment variation of the estimated x_iVp_j's as a way for estimating volatility. The evaluation team is asked to record their judgment of possible variation, $\pm$ *% var*, of the previously estimated x_iVp_j's. A *+%var* corresponds to an anticipated percentage increase in the x_iVp_j. A *-%var* corresponds to an anticipated percentage decrease in the x_iV_{pj}. Possible *%var* values may be then

available for the optimistic, the pessimistic, and likely x_iV_{pj}'s respectively given by *Optimistic* $x_iVp_j \pm \%var_o$, *Likely* $x_iVp_j \pm \%var_l$, *Pessimistic* $x_iVp_j \pm \%var_p$. In real options, σ calculates to the standard deviation of the rate of return on the asset. Intuitively, the *%var* is analogous to the rate of return on the architectural potential. Accordingly, we take the percentage of the standard deviation of the x_iVp_j variation estimates-the optimistic, likely, and pessimistic values to calculate σp_j.

Construct call options to calculate the option relative to this valuation point of view

Having estimated the major parameters of the model, it is now possible to compute the call options using (5.2) and (5.3) on the architecture in supporting change i. As we have noticed, several estimates for $C_{ei}p_j$ and x_iVp_j, ranging from optimistic to pessimistic or representing possible valuation scenarios, would have been computed at the end of the valuation and relevant to a valuation point of view P_j. Examples are depicted in Table 5.4. Based on the case and the evaluation objectives, the analyst may then compute optimistic, pessimistic, or likely options.

Table 5.4. Example of estimated parameters at the end of the valuation

Variable	Estimated Parameters
$C_{ei}p_j$	*Optimistic* $C_{ei}p_j$
	Likely $C_{ei}p_j$
	Pessimistic $C_{ei}p_j$
x_iVp_j	*Optimistic* x_iVp_j
	Likely x_iVp_j
	Pessimistic x_iVp_j
σ_{pj}	*Optimistic* $x_iVp_j \pm var_o$
	Likely $x_iVp_j \pm var_l$
	Pessimistic $x_iVp_j \pm var_p$

$$E\,[\max\,(x_i Vp_j - C_{ei}p_j,\, 0)] \qquad (5.2)$$

$$C = x_i Vp_j\, N\,(d_1) - C_{ei}p_j e^{-r\,(T)}\, N\,(d_2) \qquad (5.3)$$

where,

$$d_1 = \frac{\ln(x_i Vp_j \,/\, C_{ei}p_j) + (r + \sigma_{pj}{}^2/2)(T)}{\sigma_{pj}\,(T)^{1/2}}$$

$$d_2 = \frac{\ln(x_i Vp_j \,/\, C_{ei}p_j) + (r - \sigma_{pj}{}^2/2)(T)}{\sigma_{pj}\,(T)^{1/2}} = d_1 - \sigma_{pj}\,(T)^{1/2}$$

For numerical examples, we refer the interested reader to Chapter 6, mainly to Sections 6.2.3, 6.2.4, 6.3.6.2, 6.3.6.3, and 6.3.7 where we show how (5.2) can be applied and how the relevant parameters could be estimated in the context of use.

5.3 Phase III. Interpretations and Recommendations

The final stage of the method is the evaluation and the interpretation of the results relative to the set objectives. The supporting method is open and flexible enough to address many evolution-related objectives. The method does not define rigorous or prescribed actions to follow. Although the steps are numbered suggesting linearity, this is not a strict waterfall process. There were be times when an analyst will return briefly to an earlier step; will jump forward to a later step; or will iterate among steps, as the need dictates. Furthermore, the analyst may amend the steps, based on the available information at hand, the case itself, and the set evaluation objective(s). Accordingly, the nature of the decisions due to the application of the model fundamentally varies with the nature of the problem, across projects, and organizations. As a result, such decisions are subject to the objective for which the model/method is applied. In chapter 6, we will explore how the computed options value (i.e., the options-in-the-money or the options-out-of-the-money) may be used to provide insights into architectural stability and investment decisions related to the

evolution of the software. In a nutshell, the recommendations are tailored to the set evaluation objective(s). The computed options values guide the recommendations. Below, we explore some dimensions that the recommendations may address:

- **Trade-off analysis.** The evaluation may aim at comparing two or more architectures and select one, which is likely to be stable in the face of some probable critical future changes in requirements. In this context, the application of the model has to explore points where the candidate architectures is in-the-money or out-of-the-money to inform the trade-off analysis and steer subsequent recommendations. Interested reader may refer to the case of selecting a "more" stable induced-middleware architecture, presented in chapter 6, for an example.

- **The worthiness of reengineering or designing the architecture for change and its stability implications.** The evaluation may aim at assessing the worthiness of investing in reengineering or designing the architecture for the change and its stability implications. In this context, the application of the model has to explore situations where investing in such an exercise may add a value to the software system and/or the enterprise owning the architecture. Again, the value of the computed calls provide the analyst with insights into when it might be cost-effective to invest in such an exercise, while not sacrificing the available resources. Accordingly, related recommendations on the cost-effectiveness of such an exercise, its long-term value, and its stability implications may follow. Interested reader may refer to the refactoring case of chapter 6, for an example.

- **Flexibility of the architecture relative to critical changes in requirements and its stability implications**. The evaluation may aim at identifying critical change(s) for which the software architecture is likely to be inflexible. These changes may exhibit future threats on the stability of the architecture of the software system. In this context, the computed call options may provide insights into probable risks, technical risks or investment-related, that could confront the architecture during its lifetime. The risk could be attributed, for example, to the problematic architectural decisions, the limitations of the

existing infrastructure, and/or the inflexibility of the architectural style in accommodating the likely future critical change in requirements.

- **Others: Strategic "performance" of the architecture over time: Success (failure) of evolution.** The evaluation may aim at examining the extent to which the architecture can support future growth and unlock future opportunities, such as extending the range of services while leaving the architecture intact, or instantiating from the core architecture new market products. In this context, the architecture is the appropriate level of abstraction at which to think of strategic software decisions and guide the evolution of the software system. The computed call options may provide an insight into the success (failure) of evolution and the "performance" of the architecture over time through sustaining evolution and generating value. Recall, software evolution need to seek and create value relative to the resources invested [Bahsoon and Emmerich, 2004b]. As such, the costs of evolving software should not outweigh the returns from the process to achieve a net benefit. The future net benefits are very much correlated to the extent to which the architecture can "pull" the options. When the call options are in-the-money, then this is a suitable measure for the "resilience" of the architecture to change and the success of evolution. When the call options are out-of-the-money, then this is indicative to either the over flexibility of the architecture (e.g., waste of recourses), unutilized flexibility, or inflexibility of the architecture while achieving its evolutionary milestones. Accordingly, the situation and the options results may steer subsequent strategic recommendations.

5.4. Summary

In this chapter, we have described a three-phase method for conducting an architectural evaluation for stability using ArchOptions. We have discussed issues related to conducting these steps, as it was realized in the application of ArchOptions. The method does not prescribe rigorous steps to follow upon using ArchOptions; it aims to discuss issues and provide ways for estimating the ArchOptions parameters.

We have provided guidelines on eliciting the likely changes in requirements and relating the change to architecture. For valuing the flexibility of an architecture to change, we have outlined a valuation points of view framework. The framework is flexible enough to account for the economic ramifications of the change on the structural (e.g., maintainability) and behavioral (e.g., throughput) qualities of an architecture and the associated business goals (i.e., market). The framework can incorporate multiple valuation techniques, some with subjective estimates, and others based on market data, when available. We have explored ways for estimating the ArchOptions parameters in the context of use.

In chapter 6, we will explore cases that highlight possible application of the model and its supporting method.

Chapter 6

Evaluation - Applying ArchOptions

In previous chapters, we have described a model for predicting architectural stability. We have supported the model with a three-phase method. In this chapter, we report on our experience in using the model and its supporting method on two case studies.

6.1 The Evaluation Method in Brief

Case studies have been extensively used to empirically assess software engineering approaches [Maciaaszek and Liong, 2004]. When performed in real situations, case studies provide practical and empirical evidence that a method is appropriate to solve a particular class of problems. According to Dawson [Dawson et al., 2003], conducting controlled and repeatable experiments in software engineering is quite difficult, if not impossible to accomplish. This is mainly because the way software engineering methods are applied varies across different contexts and involve variables that cannot be fully controlled. Nonetheless, we consider that case studies are the most appropriate approach to evaluate "soft" methods like ArchOptions. The DESMET methodology [Kitchenham et al., 1997] provides hints for guiding the evaluation of software engineering methods. The authors state that the first decision to make when undertaking a case study is to determine what the study aims to investigate and evaluate. For evaluating ArchOptions with case studies, we aim at evaluating the thesis in the large and in the small, as detailed below:

Evaluation of the thesis in the large aims at exploring the approach "fitness" in addressing representative architecture-centric evolution problems, with desired stability requirements. The evaluation aims at demonstrating the approach's applicability, simulating the model's application, evaluating the maturity of the model's interpretations, and highlighting possible insights that could derive from the model's application to said problems. In the first case study, we explore how ArchOptions can be used to assess the worthiness of re-engineering a "more" stable architecture in the face of likely future changes in future requirements. We take refactoring as a representative example of reengineering. In the second case study, we show how ArchOptions can inform the selection of a "more" stable middleware-induced software architecture in the face of future changes in non-functional requirements, such as changes in scalability requirements. As part of the evaluation, we argue that ArchOptions is well suited to address these architecture-centric to evolution problems.

Evaluation of the thesis in the small aims at extending the confidence in the following specific claims:

- The uncertainty, attributed to the likelihood of change(s), makes real options theory superior to other valuation techniques, which fall short in dealing with the value of architectural flexibility under uncertainty. For some examples, we compare the options results to other valuation techniques.
- The flexibility of an architecture in face of likely changes in requirements creates values in the form of real options.
- The problem of finding a potentially stable architecture requires finding an architecture that maximizes the yield in the added value, relative to some likely future changes in requirements. If we assume that the added value is attributed to flexibility, the problem becomes maximizing the yield in the embedded or adapted flexibility in a software architecture relative to these changes.
- The decision of selecting a potentially stable architecture has to maximize the value added relative to some valuation points of view: we demonstrate the use of the valuation points of view framework in capturing the options on an evolving architecture from different perspectives and informing the selection.

We use representative examples from the above-mentioned case studies to empirically extend the confidence in these claims. Though some of these examples are conducted in controlled environments, they are adequately representative of analysis and decisions taken in real small to medium scale projects.

We evaluate ArchOptions on some qualitative characteristics including simplicity of use, prediction effectiveness, computation correctness, openness, and comprehensiveness. We reflect on ArchOptions strengths and limitations upon conducting the case studies.

When sufficient information is available, we relate the conducted case studies steps to that of the method sketched in Chapter 5. Nevertheless, the case studies are structured in a way that could ease future replication.

We discuss some observations and conclusions that have derived from the case studies. These could either relate to the application of the approach itself and/or reflect on the practical and proactive understanding of the architectural stability problem as observed when conducting these cases.

6.2 Applying ArchOptions to Value the Payoff of Refactoring

In this section, we use ArchOptions to value the payoff of investing in a refactoring exercise [Bahsoon and Emmerich, 2004a; Bahsoon and Emmerich, 2004b]. The valuation is based on a tradeoff between the upfront investment in refactoring and the future benefits, due to the enhanced structural flexibility resulting from this exercise.

In subsequent sections, we motivate the need for valuing the payoff of refactoring using ArchOptions, in the absence of suitable models for such a valuation. We apply ArchOptions to a refactoring case study from the literature [Leitch and Stroulia, 2003]. We discuss the rationale of the case study. We report on the results of the ArchOptions application. In more abstract terms, the case study shows how

ArchOptions can be applied to assess the worthiness of re-engineering to obtain a "more" flexible structure, which has better prospect in accommodating likely future changes in requirements. Research wise, the case demonstrates a novel application of real options theory to the valuation of the payoff of refactoring [Bahsoon and Emmerich, 2004b].

6.2.1 Motivation

As software is enhanced, modified, or adapted to new requirements, the software becomes more complex and drifts away from its original design. To reduce complexity, there is a need for techniques that incrementally improve the internal software quality. The research domain that addresses this problem is referred to as restructuring, or in the case of object-oriented and agile development, as refactoring [Mens and Tourwe, 2004]. In the context of software evolution, restructuring and refactoring are used to improve the quality of the software such as extensibility, modularity, reusability, complexity, and efficiency. Refactoring refers to the process of changing an (object-oriented) software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [Mens and Tourwe, 2004]. In refactoring, the key idea is to redistribute classes, variables, and methods across the class hierarchy in order to facilitate future adaptations and extensions. This in turn will result in a modified structure (compared to the original) with different qualitative measures and value potentials.

Numerical measures can be used before refactoring, to measure the quality of software, or after the refactoring, to measure improvements of the quality. For example, Simon et al. [2001] use distance-based cohesion metrics to detect where in a given piece of software there is a need for refactoring. Kataoka et al. [2002] use coupling metrics to evaluate the effect of refactoring on maintainability. Coleman et al. [1994] use a polynomial of multiple measures to define a maintainability index by means of which the effect of refactoring can be evaluated. However, little work has been done on understanding the economics of refactoring. For example, when is it cost-effective to invest in a refactoring exercise? How can we value the payoff due to refactoring, prior to investing in such an exercise? How can we reason about this payoff in connection with changes in the structure and at correspondingly higher

level of abstractions than code? These questions translate into a need for economic models that quantify the payoffs of refactoring. Such models inform the decision in investing in refactoring through a tradeoff between the up-front cost and the expected added value to the system as a result. The added value may be strategic or operational; it may take the form of expected savings in maintenance and/or returns due to the enhancement of some qualities such as reusability or efficiency. A characteristic of these benefits, whether strategic or operational, is that their payoffs are uncertain and may not be immediate.

Notable effort on understanding the economics of restructuring and refactoring includes [Leitch and Stroulia, 2003; Sullivan et al., 1999]. Leitch and Stroulia [2003] have proposed a framework for predicting the return on investment (ROI) for a planned refactoring using cost-benefit analysis. Sullivan et al. [1999] have shown how options thinking can be used to value software design decisions including restructuring. They have developed an option model that borrows from decision analysis to value the payoff of the decision to restructure legacy systems and its optimal exercise time.

6.2.2 The Case Study Rationale

Refactoring a system enhance the flexibility of the system's structure/architecture. Yet, this incurs an upfront cost to investment. It is worth investing in refactoring, if the refactored system could lead to an architecture/structure that is more flexible and adds a value to the system or the enterprise following this exercise. We use the expected benefits, due to the enhanced flexibility in the structure, as a way to value the payoff of refactoring. As the added value is attributed to the enhanced flexibility in the structure, the decision to refactor is driven by the motivation to maximize the payoffs in the adapted architectural flexibility that results from refactoring. We use future savings in maintenance costs, relative to some likely future changes, as a way to quantity the added value.

We apply ArchOptions to a refactoring case study from the literature [Leitch and Stroulia, 2003]. The objective of the study is to empirically simulate the applicability of the model and validate its interpretations. We summarize the simulation rationale as follows: (a) refactor and observe its effect on the flexibility of the structure (b) observe the potential of the structure to some random changes in requirements; (c) quantify flexibility relative to likely future changes as a way for understanding the payoff of refactoring. Particularly, we seek an understanding for the following: Are the model interpretations valid? When does refactoring, as an adapted flexibility, add to the system a value? How valuable is it investing in a refactoring?

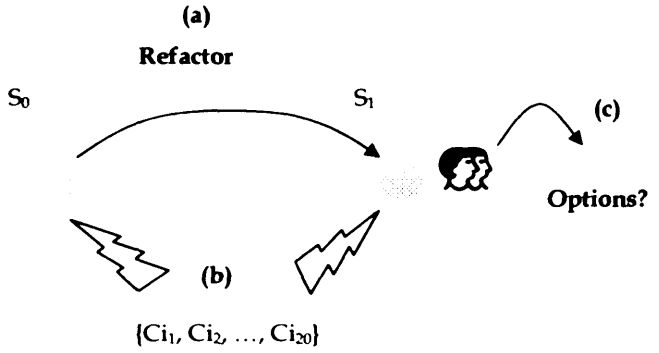


Figure 6.1. Sketch of the simulation rationale

To achieve the simulation rationale, we use the refactoring case study of a traffic light system published in [Leitch and Stroulia, 2003]. Leitch and Stroulia [2003] propose a framework to predict the return on investment (ROI) for a planned refactoring using cost-benefit analysis. We recast the problem into an option problem: we consider the benefits of refactoring to be uncertain as the demand for future changes -following refactoring- are uncertain. We restrict architectural information to data and control dependency for this case. Table 6.1 summarizes the structural changes upon evolving S_0 (the initial structure) to S_1 (the refactored structure) of the traffic light system. Table 6.1 shows that refactoring has transformed the structure into a more flexible state through the decrease of both control and data dependencies. The decrease in dependencies in S_1 means less complexity, better prospects for accommodating

future changes, and better potential for maintenance savings [Mansour and Bahsoon, 2002].

Table 6.1. Aggregate results: the change (%) - evolving S_0 to S_1

Size in SLOC	740	602	-19%
No. of Modules	29	38	31%
Avg. SLOC Per Module	26	16	-38%
Data Dependency	147	112	-23.60%
Control Dependency	101	73	-19.40%

6.2.3. Valuing the Payoff of Refactoring

Refactoring, a preventive change, can be seen as an investment to embed flexibility. The objective is to "clear up" much of the degraded system structure and enhance its upside potentials by making it more accommodating for future changes. In this context, refactoring can be seen as an investment to purchase growth options that enhance the upside potentials of the structure, paying an upfront cost I_e, which corresponds to the cost of refactoring. We build on the ArchOptions model to value whether it is worthwhile to invest into refactoring, as shown in (6.1):

$$payoff = V_{Dev} - \sum_{i=0}^{n} I_e + \sum E\,[\max\,(x_i V - C_{ei}, 0)] \qquad (6.1)$$

Let us assume that S_1 is a structure of the software obtained by refactoring S_0. We assume that refactoring is an economical choice, if it adds value to S_1 relative to S_0. We attribute the added value to the enhanced flexibility of S_1 over S_0. If we are considering savings in maintenance as a criteria for understanding the value added to the system, then future changes in requirements following refactoring will tell us how valuable S_1 is relative to S_0. But the added value due to refactoring is uncertain, as the demand on future changes are uncertain. This makes refactoring a good candidate to reason using option "thinking".

The decision to refactor has to be guided by the expected payoff in $(- I_e + \sum_{i=1...n} E\,[\max\,(x_i V - C_{ei}, 0])_{S1}$ relative to that of S_0. That is, if $(- I_e + \sum_{i=1...n} E\,[\max\,(x_i V - C_{ei}, 0)]_{S1}$

> $\sum_{i=1...n} E\ [\max\ (x_i V - C_{ei}, 0)]_{S0}$) for some likely changes, then it is worth investing in refactoring, as the investment is likely to generate more growth options for S_1 than for S_0. As we assume that $x_i V$ is the expected saving in S_1 over S_0 due to refactoring, it is reasonable to consider that if (- I_e + $\sum_{i=1...n} E\ [\max\ (x_i V - C_{ei}, 0)]_{S1}$ >=0), then investing in refactoring is said to payoff. An optimal payoff could be when the option value (i.e., $\sum_{i=1...n} E\ [\max\ (x_i V - C_{ei}, 0)]$ approaches the maximum relative to some changes in requirements, indicating an optimal payoff in an investment in flexibility provided that (- I_e + $\sum_{i=1...n} E\ [\max\ (x_i V - C_{ei}, 0)]_{S1}$ >= 0). The analyst may conduct sensitivity analysis to manipulate the model variables and analyze when such a situation is likely to occur.

For a requirement change k, if the (- I_e + E [max ($x_k V$ - C_{ek}, 0)]) <0, then refactoring is not likely to payoff as the flexibility of the architecture in response to the change is not likely to add a value if the change need to be exercised. Two interpretations might be possible: (i) the architecture is overly flexible in the sense that its response to the change(s) has not "pulled" the options. This implies that the embedded flexibility (or the resources invested in implementing flexibility) are wasted and unutilized to reveal the options relative to the changes. In other words, the degree of flexibility provided is much more than the flexibility demanded for the change. This case has the prospect in providing an insight on how much we need to invest in refactoring relative to the likely future changes, while not sacrificing much of the resources; (ii) the other case is when the architecture is inflexible relative to the change. This is when the cost of accommodating the change is much more than the cumulative expected value of the architecture potential relative to the changes.

We apply the model: we construct a call option for the likely changes following refactoring. To capture and estimate $x_i V$, we restrict the valuation to the development perspective. We use the expected savings in development effort for likely futures changes due to refactoring. When necessary, we use $2000 for man-month to cast the effort into cost. We show how we have estimated the parameters:

Estimating (I_e). Table 6.3 reports the refactoring effort (man-month), cost ($), and schedule (month) based on the refactoring plan presented in [Leitch and Stroulia, 2003] and given in Table 6.2. Table 6.3 provides three values: optimistic, likely, and pessimistic for each parameter. All are calculated using COCOMO II.

Table 6.2. The proposed refactoring plan and its design impact [Leitch and Stroulia, 2003]

Proc. No.	Refactoring	Add SLOC	Del. SLOC	Proc. No.	Refactoring	Add.	Del.
1	Extract Method	24	225	33	Extract Method	27	0
2	Extract Method	4	28	34	Extract Method	81	0
10	Move Method	4	49	35	Extract Method	17	0
11	Extract Method	4	56	36	Extract Method	9	0
30	Extract Method	4	0	37	Move Method	13	0
31	Extract Method	9	0	38	Extract Method	14	0
32	Extract Method	10	0	-	-	-	-
	SUBTOTAL:	59	358		SUBTOTAL:	161	0
					TOTAL:	220	358

Table 6.3. Refactoring effort, schedule, and cost

	Effort			Schedule			I_e		
	Op	Lik	Pes	Op	Lik	Pes	Op	Lik	Pes
Refactoring	0.9	1.2	1.5	3.6	3.9	4.2	1893	2366	2958

Estimating (x_iV). To value the architectural potential of S_1 due to refactoring, we use twenty changes to stress S_1 with cost given as C_{ri}. The twenty changes are of an adaptive nature; they are generated based on percentage estimates of design, integration, and code to be modified per change. Examples of these changes includes

adding/deleting a functionality in the Traffic Light system, integrating with other systems, enhancing the functionality, etc. The same likely changes were used to stress S_0. The objective is to calculate the difference (i.e., savings-if any) in effort/cost of S_1 over S_0. The aim is to quantify the architectural potential due to the embedded flexibility, from the development perspective. We use COCOMO II to estimate the effort/cost for the twenty changes on each structure. x_iV corresponds to the difference- as reported in Table 6.4. Expected savings, due to refactoring, are in the range of $12806 (optimistic) to $7433 (pessimistic) for the twenty changes.

Calculating volatility (σ). The volatility of the stock price (σ) is a statistical measure of the stock price fluctuation over a specific period of time; it is a measure of how uncertain we are about the future of the stock price movements. Volatility stands for the "fluctuation" in the value of the estimated x_iV. Intuitively, it "aggregates" the "potential" values of the structure in response to the change(s). To calculate σ, we follow the real options principles to calculation taking the percentage of the standard deviation of some representative estimates of x_iVs over a period of interest. In some cases and for the sake of simplicity, we use three estimates of the x_iVs: these are optimistic, likely, and pessimistic values.

Exercise time (t) and free risk interest rate(r). As a simulation assumption, we set the exercise time to three years. We set the free risk interest rate to zero (i.e., assuming that the value of money today is the same as that in three years time).

6.2.4 Results and Discussion

Below, we discuss the results of applying ArchOptions to value the payoff of refactoring.

Observation 1. Flexibility creates real options: S_1 is more flexible than S_0 (due to decrease in dependencies as a result of refactoring); S_1 has created more real options when compared to S_0.

Tables 6.4 and 6.5 shows that S_1 is in the money in response to the twenty random changes- relative to the development perspective. The results indicate that refactoring (i.e. as the embedded flexibility in S_1) is likely to enhance the option value by \$5979 (pessimistic) to \$10593 (optimistic) over S_0, if the twenty changes need to be exercised following refactoring. Thus, as flexibility is improved, S_1 is likely to add value in the form of options in response to the twenty changes.

Table 6.4. Options on S_1 relative to S_0 (\$) for the twenty likely changes (Maintenance valuation point of view)

	Pessimistic			Likely			Optimistic		
	C_{e1}	T	x_1V	C_{e1}	T	x_1V	C_{e1}	T	x_1V
	1454	3	7433	1817	3	9292	2212	3	12806
Option	5979.09			7474.6			10593		

Table 6.5. Options on S_1 for one to ten changes at a time

		NV			Options		
Changes	σ	Pes.	Lik.	Op.	Pes.	Lik.	Op.
1Req.Ch.	1.4	371.7	464.6	640.3	0	0	0
2 Req.Ch.	2.7	743.3	929.2	1280.6	0	0	0
3 Req.Ch.	4.1	1115.0	1393.8	1920.9	0+	0+	1.2
4 Req.Ch.	5.5	1486.6	1858.4	2561.2	73.6	92.45	334.9
5 Req.Ch.	6.8	1858.3	2323.0	3201.5	405.6	507.6	989.07
9 Req. Ch.	12.2	3339	4181.4	5760	1885	2364	3547
10 Req.Ch.	13.6	3717	4640	6400	2263	2823	4188

Observation 2. How valuable is refactoring?

Let us take the average value of the twenty changes. The objective is to simulate the responsiveness of S_1 to one likely average change. The result of Table 6.5 implies that though S_1 is flexible, refactoring has not "pulled" the options for one change. S_1 is said to be out of the money for this change. This implies that the embedded flexibility (or the resources invested in implementing flexibility) are wasted and unutilized to reveal the options relative to this change. In other words, the degree of flexibility provided is much more than the flexibility demanded for this change. We repeat the above experiment, but stressing S_1 with two, three, four, and then five average changes at a time. Using two average likely changes, the options reported zero values. Again, two likely average changes have not "pulled" the options. Interestingly, S_1 has just about pulled the options for three changes. For four, five, and nine changes, S_1 reveals the options; however, refactoring is not likely to payoff as $(- I_e + \sum_{i=1...n} E [\max (x_i V - C_{ei}, 0)]_{S1} < 0)$. For ten changes, refactoring is expected to payoff as $(- I_e + \sum_{i=1...n} E [\max (x_i V - C_{ei}, 0)]_{S1} > 0)$. Thus, refactoring is likely to add to the system a value, if ten or more changes need to be exercised during the next three years.

This case study has the prospect of providing an insight into how much we have to invest in flexibility to achieve stability relative to the likely future changes, while not scarifying much of our resources. In real situations, an optimal stability could be when the option value approaches the maximum, indicating an optimal payoff in an investment in flexibility. The analyst may make use of the sensitivity estimates to manipulate the model variables and analyze when such a state is likely to occur.

6.2.5 Concluding Remarks

In Table 6.6a and Table 6.6b, we relate the case study to the phases of the method described in Chapter 5. We have amended some of the steps and based on the available information at hand and the evaluation objectives. We have relaxed applying phase I, as it is assumed that the likely changes following refactoring are

provided and need not be elicited. Upon applying Phase II, we have restricted the valuation to one valuation point of view, which is the development perspective. We have appealed to the use of maintenance savings as a way to value the options due to refactoring. Needless to say, the valuation could have incorporated other valuation points of view (e.g., extensibility, reusability, efficiency) to value the options due to refactoring. Future work may entail investigating ways for valuing the payoff of refactoring relevant to other points of views. The objective is to have a comprehensive value of options from different perspectives. As for Phase III, we have reported on some observations derived from the model simulation. These are mainly on the worthiness of refactoring, as a mean for introducing flexibility into the structure. In reality, the analyst may use a similar argument to justify a case for investing in refactoring. The analyst/architect may explore situations where investing in such an exercise may add a value to the software system and/or the enterprise owing the architecture. Again, the value of the computed calls provide the analyst with insights into when it might be cost-effective to invest in such an exercise, while not sacrificing the available resources. Accordingly, related recommendations on the cost-effectiveness of such an exercise, its long-term value, and its stability implications may then follow.

Table 6.6a. Relating the refactoring case to Phase I of the method

Phase I	Case I
Setting the objectives for evaluating architectural stability	Objective: Valuing the payoff of the adapted architectural flexibility due to refactoring
Eliciting the change $\{i_1, i_2, \ldots, i_n\}$ that are critical to the set objectives	Twenty changes of adaptive type are used
Tracing the change to the architecture and its associated design decisions	Control/data flow is taken as the architectural artifacts on which the decision of the cost-effectiveness is made

Table 6.6b. Relating the refactoring case to Phase II of the method

Phase II	Case I
Estimate the cost of accommodating the change	An estimate of the cost of implementing the twenty changes
Identify the value of the architectural potentials with respect to the change	By reducing the complexity of the control/dataflow structure following refactoring, future savings in maintenance could be claimed
Identify valuation points of view	Maintainability
Volatility	Using optimistic, likely, and pessimistic

The purpose of the case study is to simulate the model steps and the maturity of its interpretations. The results demonstrate the fitness of the approach in addressing the problem of valuing the payoff of refactoring in relation to likely future changes in requirements. The observations verify that the model interpretations are reasonable. As a satisfaction of the spanning condition entailed by Black and Scholes [1973], we argue that valuation based on person-month does implicitly hold market-based data and is done in relation with the market. Alternatively, we could have cast the options model to use different options valuation (e.g., [Cox and Rubinstein, 1979]). However, the application of Black and Scholes [1973] offers a closed and an easy-to-compute solution, for it assumes that $x_i V$ is lognormaly distributed, not requiring $x_i V$ to be probability-adjusted for rise and drop in value, as when compared to [Cox and Rubinstein, 1979].

6.3 Applying ArchOptions to Select Stable Middleware-Induced Software Architectures

The current trend is to build distributed systems using *middleware*, which provide the application developer with primitives for managing the complexity of distribution and for realizing many of the non-functional requirements such as scalability and performance requirements. As non-functional requirements evolve, the "coupling" between the middleware and architecture becomes the

focal point for understanding the *stability* of the distributed software system architecture in face of change. In this case, we hypothesize that the choice of a stable distributed software architecture depends on the choice of the underlying middleware and its *flexibility* in responding to future changes in non-functional requirements. We motivate the need for an economics-driven approach to the selection of a candidate middleware that will then induce a given architecture. We draw on a case study that adequately represents a medium-size component-based distributed architecture: we report on how a likely future change in scalability requirements could impact the architectural structure of two versions, each induced with a distinct middleware: one with the Common Object Request Broker Architecture (CORBA) [Object Management Group, 2000] and the other with Java 2 Enterprise Edition (J2EE) [Sun Microsystems Inc., 2002]. We appeal to the use of two valuation points of views upon valuing the potentials of the induced-architetures in relation to likely future changes in scalability requirements. We show how we can apply ArchOptions to value the flexibility of the induced-architectures, relative to the valuation points of view, and to consequently guide the selection of a more "stable" architecture. Our hypothesis is verified to be true for the given change. We conclude the case with some observations that could stimulate future research in the area of relating requirements to software architectures.

The case study demonstrates a novel application of real options theory for informing the selection of a more "stable" middleware-induced architectures [Bahsoon et al., 2005; Bahsoon and Emmerich, 2005]. Furthermore, the observations derived upon conducting the case are likely to advance our understanding to the architectural stability problem, when addressed from practical and proactive perspective.

6.3.1 Motivation

The requirements that drive the decision towards building a distributed system architecture are usually of a non-functional and global nature [Emmerich, 2000a]. Scalability, openness, heterogeneity, and fault-tolerance are just examples. The current trend is to build distributed systems architectures with middleware technologies such as Java 2 Enterprise Edition (J2EE) [Sun Microsystems Inc., 2002]

and the Common Object Request Broker Architecture (CORBA) [Object Management Group, 2000]. Middleware simplifies the construction of distributed systems by providing high-level primitives, which shield the application engineers from the distribution complexities, managing systems resources, and implementing low-level details, such as concurrency control, transaction management, and network communication. These primitives are often responsible for realizing many of the non-functional requirements in the architecture of the software system induced. Despite the fact that architectures and middleware address different phases of software development, the usage of middleware can influence the architecture of the system being developed. Conversely, specific architectural choices constrain the selection of the underlying middleware [Di Nitto and Rosenblum, 1999]. Once a particular middleware system has been chosen for a software architecture, it is extremely expensive to revert that choice and adopt a different middleware or a different architecture. The choice is influenced by the non-functional requirements. Unfortunately, the requirements tend to be unstable and evolve over time. Non-functional requirements often change with the setting in which the system is embedded, for example when new hardware or operating system platforms are added as a result of a merger, or when scalability requirements change due to sudden increase in users, as it is the case of successful e-commerce systems [Emmerich, 2000b]. Moreover, changes in non-functional requirements are critical; they can stress an architecture considerably, leading to architectural "breakdown". The ranges in which non-functional requirements change may need to inform the selection of distributed components technology, and subsequently the selection of application server products. For example, a CORBA-based solution might meet the functional requirements of a system in the same way as a distributed component-based solution that is based on a J2EE application server. A notable difference between these two architectures will be that increasing scalability demands might be easily accommodated in the J2EE architecture because J2EE primitives for replication of Enterprise Java Beans can be used, while the CORBA-based architecture may not easily scale. The choice is not straightforward as the J2EE-based infrastructures usually incur significant upfront license costs. Thus, when selecting an architecture, the question arises whether an organization wants to invest into an J2EE application server and its implementation within an organization, or whether it would be better off implementing a CORBA solution. Answering this question without taking into

account the *flexibility* that the J2EE solution provides and how *valuable* this flexibility will be in the future relative to the likely change in load might lead to making the wrong choice. This gives rise to value the *flexibility* of the middleware-induced architecture relative to likely changes in requirements so we can understand its stability implications, as the non-functional requirements of the software system evolve.

We argue that the problem of selecting a particular middleware to induce a given architecture is an *option* problem. From the evolution perspective, the flexibility of the middleware induced-architecture in coping with changes in non-functional requirements has a value that can *assist* in predicting the stability of software architectures. More specifically, flexibility adds to the architecture values in the form of *real options* that give the right but not a symmetric obligation- to evolve the software system and enhance the opportunities for strategic growth. The added value is strategic in essence, uncertain as the demand on the future changes are uncertain, and may not be immediate. The added value may take the form of (i) accumulated savings through coping with the change without "breaking" the architecture, mostly these are changes in non-functional requirements; (ii) extending the range of services while leaving the architecture intact; and (iii) the ability to respond to competitive forces and changing market conditions that may pose higher Quality of Service (QoS) requirements, such as the demands for higher availability, scalability, reliability and so forth. From an early development perspective, given several middleware candidates, the architect has the right without the symmetric obligation to embark on a selection for inducing an architecture. A "wise" selection could be regarded as an investment to buy flexibility, which could be valued as future *growth options* [Schwartz and Trigeorgis, 2000] on the architecture of the software system. These options differ from one middleware to another.

ArchOptions has the prospect of valuing the architectural flexibility due to various types of changes in requirements. These could be functional or non-functional. However, changes in non-functional requirements are often critical and more revealing for understanding architectural stability. As the middleware realizes much of the non-functionalities, analyzing for architectural stability in the face of changes in non-functional requirements cannot be done in isolation of the middleware, for

the category of distributed system built using middleware. In this context, we tailor ArchOptions to value the growth options on the architecture to be induced relative to likely changes in non-functional requirements.

In the next sections, we describe the case study rationale. We describe how ArchOptions can be employed for understanding the value added by inducing the architecture by EJB relative to CORBA, if the change in scalability, as a representative critical change in non-functional requirements, needs to be applied.

6.3.2 The Case Study Rationale

We hypothesize that the choice of a stable distributed software architecture depends on the choice of the underlying middleware and its flexibility in responding to future changes in non-functional requirements. This is necessary to facilitate the evolution of the software system, to avoid unnecessary future investments (e.g., maintenance overhead, hardware, reverting the choice of the middleware etc.), and to ensure that future resources will be used efficiently as the requirements evolve (e.g., new servers are purchased or cycles are leased, only when necessary).

We use Duke's Bank application, an online banking application provided by Sun [Sun Microsystems Inc., http://java.sun.com], as part of the J2EE reference application. Though the study is conducted in a controlled environment, we regard the Duke's bank application to be adequately representative of a medium-size component-based distributed application. Given the software architecture of the Duke's Bank, we have instantiated from the core architecture two versions, each induced by a distinct middleware: one with CORBA and the other with J2EE. We observe how a likely future change in scalability, a representative critical change in non-functional requirements, could impact the architecture of each version. Scalability is frequently thought of in terms of numbers of users that can be supported on either a single node or collectively on all nodes in a system; it denotes the ability to accommodate a growing future load. The exact method of analyzing scalability is subject to some debate: First, the change in load demands is critical as it could impact the architecture at its various levels: structure, topology, and

infrastructure. For example, the challenge of building a scalable system is to support changes in the allocation of components to hosts without breaking the architecture of the software system, or changing the design and code of a component [Emmerich, 2000b]. Second, the change in load could impact other non-functional requirements such as performance, reliability, and availability, when the change is poorly accommodated by the middleware-induced architecture. As a result, this debate is appealing to the use of the multi-perspective valuation point of view framework we have highlighted in Chapter 5. It is appealing to the use of both the structural and behavioral valuation points of view, as depicted in Figure 6.2 and detailed below:

- On the *structural point of view*: we observe how the architecture of the given system, when induced by a particular middleware, is ready to cope or need to be maintained for supporting the change in scalability. We analyze the impact of the change by looking at the structural changes and the source lines of code (SLOC) that need to be modified/added for implementing the change, configuring, and deploying the software system. We quantify the options by looking at the cost of change on the structure of each version and by valuing the savings in maintenance, deployment, and configuration costs (if any), upon accommodating the change. We refer to this valuation point of view as the *maintainability valuation point of view*.

- On the *behavioral point of view:* we use *throughput* or the capacity of the system to measure scalability. Throughput is a performance criterion, which expresses the amount of work performed by the system under test during a unit of time. We refer to this valuation point of view as the *throughput valuation point of view*.

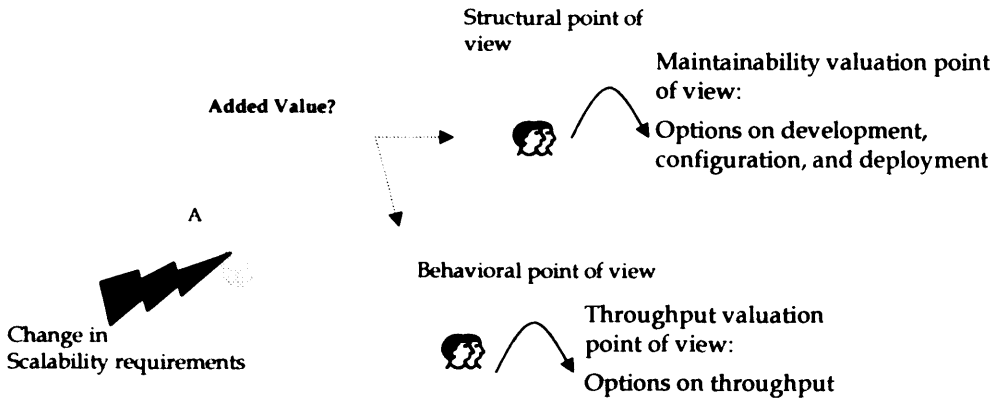


Figure 6.2. The use of structural and behavioral valuation points of view to capture the options on the induced-architecture, A, for a likely change in scalability

Hence, the ability to scale the software of a given architecture is rich for analyzing the architectural stability problem, as the change have both structural and behavioral impacts. The objective of the case is to demonstrate how structural and behavioral impact analysis on a system of a given architecture can be complemented with options "thinking". The rationale is that by complementing structural and behavioral impact analysis with value-based calculation, the combination could provide the architect/analyst with a useful tool for understanding the extent to which the software system is flexible to accommodate likely future change in scalability requirements. The combination can provide insights on the likely success (failure) of software evolution, and consequently on the potential stability of the architecture to change. This combination can also provide cost and value indictors of the impact of the change on the structure and the behavior of the system. For example, throughput and performance are correlated with value. That is, the more business transactions can be performed on a system of a given architecture, the more value is said to be created for the enterprise. Therefore, "hurting" the performance of the software, upon accommodating the change in scalability requirements, implies "hurting" value.

We describe how ArchOptions can be used to inform the selection of potentially more stable middleware-induced software architectures and relative to the two valuation points of view. ArchOptions is applied to account for both the long-term value and cost of the architectural potential to the change on each valuation point of view. Given several middleware candidates, the application of ArchOptions aims at informing the trade-offs and consequently the selection of a middleware-induced architecture through a simple and intuitive calculation. Questions of interest, however, are: how valuable is the flexibility of either alternative, relative to the valuation point of view, will be in the long-run? How can we decide which solution is of a better long-term value? How can we inform the selection of a "more" stable middleware-induced infrastructure, which maximizes the yield in the added value relative to the change and the valuation points of view? For instance, the ranges in which the throughput requirements change and their value implications may need to inform the selection. At the same time, the cost-effectiveness of maintaining the structure to realize the change is another important factor. Hence, the economic interplay between evolving requirements, relative to the valuation points of view, and architectural stability needs to be addressed.

6.3.3 Setting

The architecture of the Duke's Bank application is given in Figure 6.3. The Duke's Bank has two clients: an application client used by administrators to manage customers and accounts and a Web client used by customers to access account statements and perform transactions. The server-side components perform the business methods: these include managing customers, managing accounts, and managing transactions. The clients access the customer, account, and transaction information maintained in a database.

The CORBA version of the Duke's Bank is a straightforward implementation of the above description. In the J2EE, the application consists of six EJB (Enterprise Java Beans) components that handle operations issued by the users of a hypothetic bank. The six components can be associated with classes of operations that are related to bank accounts, customers and transactions, respectively. For each of these classes of operations, a pair of session bean and entity bean is provided. Session beans are

responsible for the interface towards the users and the entity beans handle the mapping of stateful components to underlying database table. The EJBs that constitute the business components are deployed in a single container within the application server, which is part of the middleware.

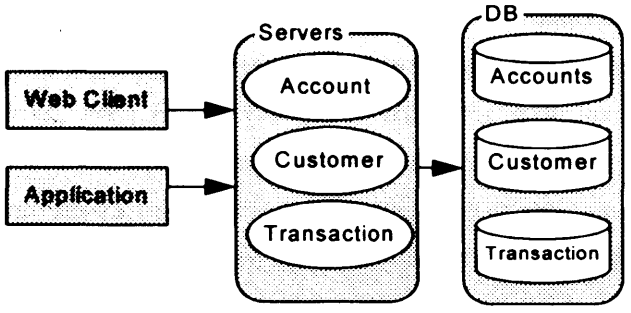


Figure 6.3. The architecture of the Duke's Bank

For the J2EE version, we use JBoss application server [http://www.jboss.org], an open source. In one of the studies, we use WebLogic server [http://www.bea.com/] with an average upfront payable license cost equal to $25000/host. We use JacORB, version 2.0 to implement the CORBA version. JacORB, is a CORBA implementation written in Java; it allows the communication of Java objects. Our choice of JacORB makes the comparison between the two versions feasible and meaningful, as both will be implemented in JAVA.

We assume that the Duke's Bank system is likely to "scale up" to accommodate a growing number of clients in a year time. As we have mentioned before, we observe how a likely future change in scalability requirements, a representative critical change in non-functional requirements, could impact the architecture of the middleware-induced architectures. We look at two valuation points of view to understand the likely impact on the architecture. For the *maintainability point of view*, we elicit the scalability "primitives" which need to be implemented or need to be maintained for scaling the structure. We analyze the impact of the change on each middleware-induced architecture. For the *throughput point of view*, we elicit the likely ranges in future load. We then discuss the impact of likely change in future load on

the behavior (throughput) of the system. In next sections, we deal with each of the above views separately.

6.3.4 The Maintainability Valuation Point of View

We consider the *Maintaining the structure for scalability* as a goal that needs to be achieved by the architecture of the software system to be induced. Following the method of Chapter 5, we adopt a goal-oriented approach to refining requirements (e.g., [Dardenne et al., 1993; Anton, 1996]). We refine the goal, using guidance on how it could be operationalized by the architecture, when induced by a particular middleware. In more abstract terms, the guidance was given through the knowledge of the domain; vendor's specification, such as [Object Management Group, 1999-2000; Sun Microsystems Inc., 2002]; related design and implementation experience, mainly that of [Othman et al., 2001a; Othman et al., 2001b]. We note that different architectural mechanisms may operationalize the this goal. As an operationalization alternative, we use replication as way for maintaining scalability on the structures. The reason is due to the fact that both CORBA and J2EE do provide the primitives or guidelines for scaling a software system using replication, which make the comparison between the two versions feasible. In particular, the Object Management Group's CORBA specification [Object Management Group, 1999-2001] defines a fault tolerance and a load balancing support, both when combined provide the core capability for implementing scalability through replication. Similarly, J2EE provides the primitives for scaling the software system through replication. Hence, the refinement and its corresponding operationalization are guided by the solution domain (i.e., the middleware). Refinement of the scalability goal is depicted in Figure 6.4. Detailing the refinements and the operationalization of the goal is given in subsequent sections.

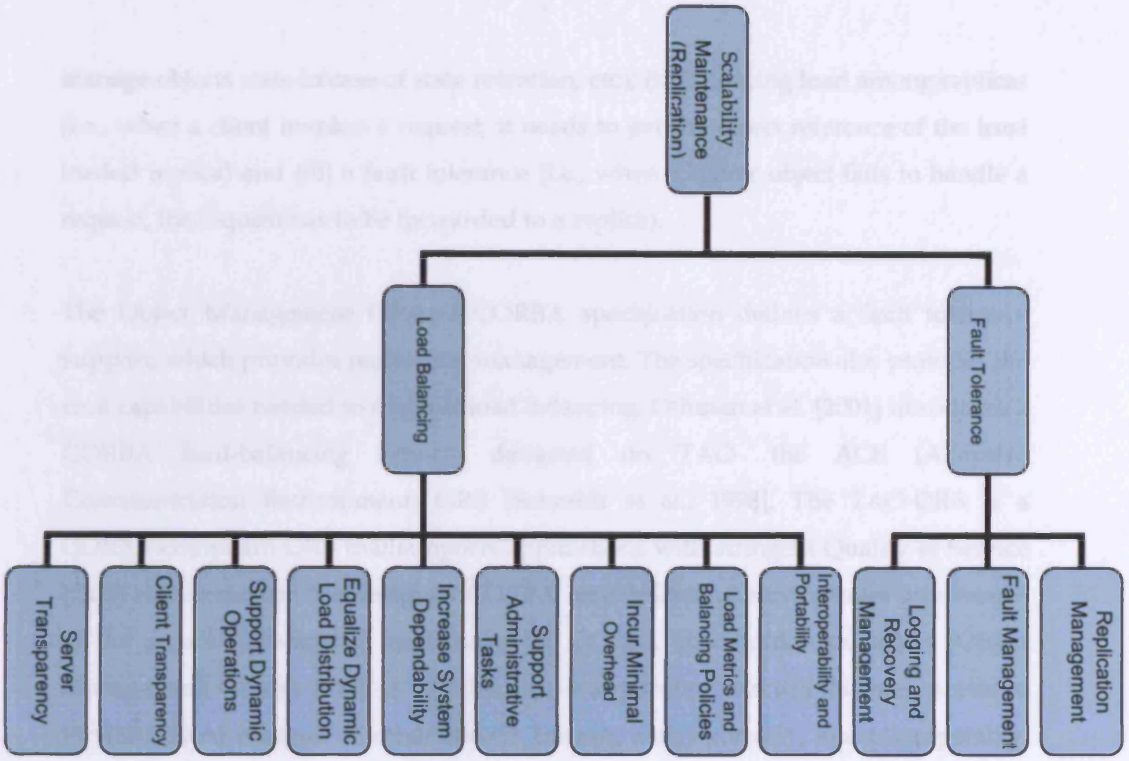


Figure 6.4. The Goal-oriented (high-level) refinement for achieving scalability through replication

6.3.4.1 Scaling the CORBA-Induced Architecture

In this subsection, we investigate how scalability could be achieved in the CORBA-induced version through replication mechanisms. The objective of this subsection is to detail the refinement of the goal (*Maintaining the structure for scalability)* and in relation to the structure to be induced.

CORBA's object model [Object Management Group, 2000] relies to a large degree on the semantics of *object references*. An object reference uniquely identifies a local or remote object instance- clients can only invoke an operation on an object if they hold a reference to the object. Managing scalability in CORBA, through replication, is not straightforward, for object referencing makes it demanding. If several replicas of a server object are available, providing an object reference to the client is uneasy task. A CORBA implementation to the management of scalability, through replication, has to incorporate the following: (i) Replication management (i.e., create, remove,

manage objects state in case of state retention, etc); (ii) balancing load among replicas (i.e., when a client invokes a request, it needs to get the object reference of the least loaded replica) and (iii) a fault tolerance (i.e., when a server object fails to handle a request, the request has to be forwarded to a replica).

The Object Management Group's CORBA specification defines a fault tolerance support, which provides replication management. The specification also provides the core capabilities needed to support load balancing. Othman et al. [2001] introduces a CORBA load-balancing service, designed on TAO- the ACE (Adaptive Communication Environment) ORB [Schmidt et al., 1998]. The TAO-ORB is a CORBA-compliant ORB that supports applications with stringent Quality of Service (QoS) requirements. The designed CORBA load-balancing service takes advantages of the request forwarding mechanism the CORBA specification mandates [Object Management Group, 1999]. A CORBA server application can use this mechanism to forward client requests to other servers transparently, portably, and interoperably. The combination of the CORBA fault tolerance support and Othman's CORBA load-balancing service provides a strong example of implementing scalability in CORBA. We use both the Object Management Group's CORBA specification and the TAO's design and implementation of the services as guidelines for understanding the structural impact of the change on the Duke's Bank architecture and the corresponding effort/cost required to scale the system.

In the below subsections, we describe the requirements and the architecture for implementing fault-tolerance in CORBA, based on the OMG specification [Object Management Group, 1999]. We describe the requirements and the architecture for implementing the load-balancing support in CORBA, based on [Othman et al., 2001a; Othman et al., 2001b]. We analyze the structural impact, when the fault-tolerance and the load-balancing services need to be implemented to scale the CORBA-induced Duke's Bank architecture.

Maintaining fault tolerance support and replication management

This subsection relates to how the *Maintaining Fault Tolerance* (subgoal of Figure 6.4) is refined and operationalized.

The Fault Tolerant CORBA standard provides robust support for applications that require a high level of reliability, beyond the level provided by single backup server. To render an object fault-tolerant, several replicas of the object are created and managed as an *object group*. Because of the object group abstraction, the client objects are not aware that the server objects are replicated (*replication transparency*) and are not aware of faults in the server replicas or of recovery from faults (*failure transparency*). The standard provides support for fault detection, notification, and analysis for the object replicas. The standard also supports a range of fault tolerance strategies, including automatic check pointing; logging and recovery from faults; request retry, and redirection to an alternative server.

The requirements for implementing Fault Tolerance in CORBA are depicted in Table 6.7 and detailed in the CORBA fault tolerance specification of the OMG [Object Management Group, 1999].

Table 6.7. The requirements for implementing fault tolerance in CORBA

Replication Management	Property Manager	Provide operations that set properties for object groups
	Object Group Manager	provide operations that allow an application to exercise control over addition, removal, and obtaining the current reference and identifier locations of members of an object group
	Generic Factory	Issues requests for replicating objects (object groups), creating replicas (members of object groups), and unreplicating objects
Fault Management	Fault detection	The Fault detection component detects the presence of a fault in the system and generates a fault report
	Fault notification	The fault notification component propagates fault reports to entities that have registered for such notifications
	Fault analysis	The fault analysis component analyses a (potentially large) number of related fault reports to generate a condensed diagnosed report
Logging and Recovery Management	Logging	The Logging records the state and actions of a member of an object group in a log
	Recovery	The Recovery Mechanism sets the state of a member, either after a fault when a backup member of an object group is promoted to the primary member, or alternatively when a new member is introduced into an object group

Figure 6.5 presents an architectural strategy that realizes these requirements and fully documented in [Object Management Group, 1999]. The architecture defines minimal modifications to the application programs, existing ORBs, and for transparency to both replication and faults. These modifications allow non-replicated clients to derive fault tolerance benefits upon invoking replicated server objects. The basic blocks of the architecture are three: *Replication management; Fault Management; and Logging and Recovery Management.* Components of the Fault Tolerance Infrastructure are shown on the top of Figure 6.5. These include *Replication Manager, Fault Notifier,* and *Fault Detector.* Interested reader may refer to the Appendix B, for further details on the architecture.

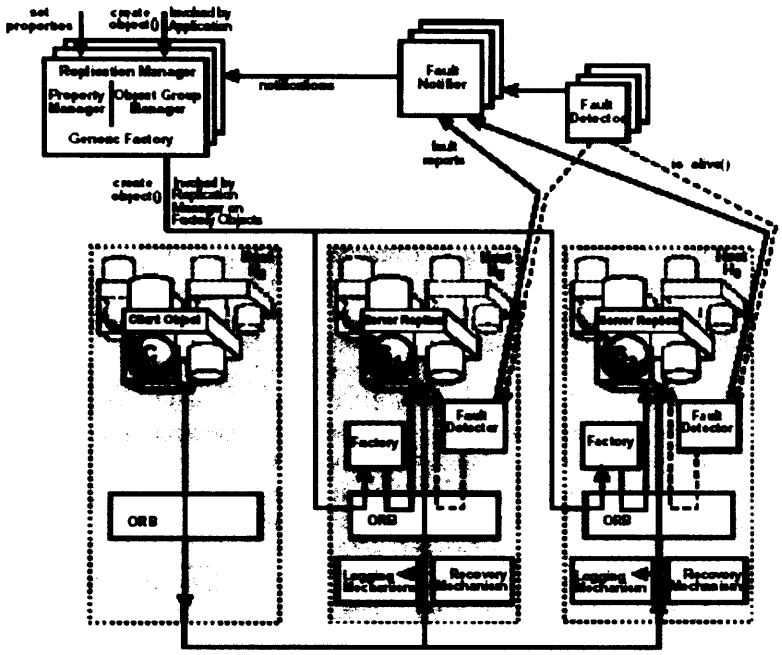


Figure 6.5. The CORBA fault-tolerance architecture [Object Management Group, 1999]

Maintaining load balancing

This subsection relates to how the *maintaining load-balancing* (subgoal of Figure 6.4) is refined and operationalized.

Load balancing helps improve system scalability by ensuring that client application requests are distributed and processed equitably across a group of servers. Likewise, it helps improve system dependability by adapting dynamically to system configuration changes that arise from hardware or software failures. According to [Othman et al., 2001a], the design of an effective CORBA load balancing service should be based on the following requirements, as depicted in Table 6.8.

Table 6.8. The requirements for Implementing load balancing in CORBA [Othman et al., 2001b]

Sub goals	Description
Enable client application transparency	A CORBA load balancing service should be as transparent as possible to clients and servers; it should require no changes to clients whose requests it balances
Enable server application transparency	Implementing a server object's servant (a programming language entity that implements object functionality in a server application) should require no changes to support load balancing. Yet changes to the server application might still be required under certain conditions
Support dynamic client operation request patterns	The CORBA load balancer, however, shall focus on load balancing techniques that do not require a priori scheduling information, where client operation request patterns are dynamic and the duration of each request might not be known in advance- which is the case of the Duke's Bank
Maximize scalability and equalize dynamic load distribution	CORBA load balancing service must enhance system scalability by maximizing dynamic resource utilization in a group of servers that otherwise would be underutilized
Increase system dependability	Load balancer should provide mechanisms to handle failures efficiently when detected by administrators or other system components. For example, the load balancer should migrate crashed or failing servers to other servers until the failure is resolved
Support administrative tasks	A good CORBA load balancing service should have facilities for dynamic addition/removal/upgrading of new replicas and should adjust to the new load conditions rapidly, without disrupting or suspending service for existing clients
Incur minimal overhead	A CORBA load balancing service should not introduce undue latency or networking, which may reduce the overall system performance
Support application-defined load metrics and balancing policies	A CORBA load balancing service should let applications specify the semantics of metrics used to measure load, such as CPU, I/O resources, communication bandwidth, or memory load
Rely on CORBA interoperability and portability	A CORBA load balancer should not restrict the application developers to single ORB providers

Othman et al. [2001b] suggest a CORBA adaptive balancing built on TAO to realize the above stated requirements. The TAO's load balancing solution is entirely based on standard features in CORBA, without requiring severe extensions to the ORB or its communication protocols. The suggested load balancing solution is based on the

patterns [Schmidt et. al., 2000] of the CORBA component model (CCM) [BEA Systems, 1999] for minimizing the changes on the application layer. In particular, the following patterns are utilized to achieve the above stated transparency requirements: these are the Portable Interceptors pattern, Component Configuration pattern, Component Configurator pattern, and the Asynchronous Completion Token pattern [Schmidt et. al., 2000]. The architecture is given in Figure 6.6. Interested reader may refer to Appendix B for technical details on the load balancing architecture.

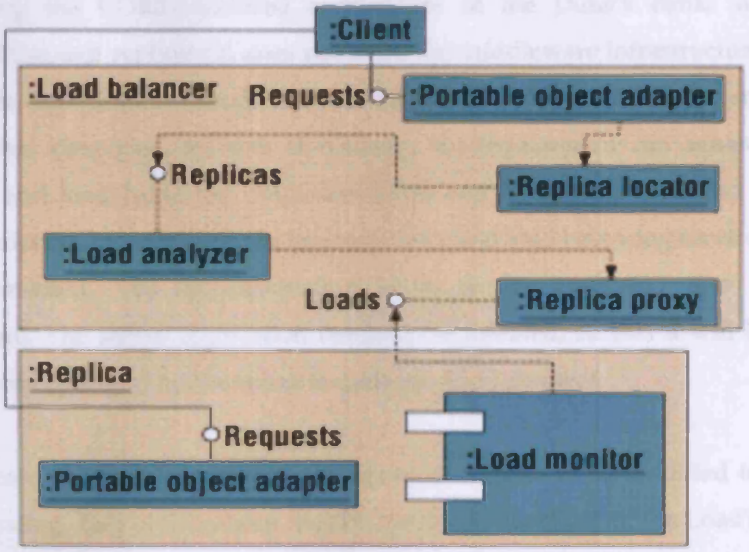


Figure 6.6. TAO load balancing [Othman et al., 2001b]

Change impact analysis

The combination of the CORBA fault tolerance support and Othman's CORBA load-balancing service provides an example on how scalability could be achieved in the CORBA-induced architectures of the Duke's Bank. In this section, we analyze the impact of the change on the Duke's Bank by looking at the structural changes and the source lines of code (SLOC) that need to be modified/added for implementing the change, configuring, and deploying the software system. We use the design and the implementation of both services (i.e., fault tolerance and load balancing) on TAO as a guide to estimate the design impact and the effort required to realize the scalability requirements in the Duke's Bank. The TAO design of these services is based on the

CORBA specification. We note that the TAO's implementation of both services is in C++. We list all the JAVA classes and files necessary to build the equivalent JAVA implementation of both services. A List of classes and files necessary to implement the fault tolerant service into the Duke's Bank architecture is depicted in Table B-1 of the appendix. Table B-2 of the appendix reports on the effort necessary to develop and integrate the load balancing service into the middleware. Table 6.9 provides an aggregated summary of the overall SLOC that need to be implemented.

Considering the CORBA-induced architecture of the Duke's Bank, supporting scalability through replication does not leave the middleware infrastructure and the application layer intact. Though the use of both CORBA specification and design patterns, has simplified the task of realizing the requirements for achieving fault tolerance and load balancing, implementation and integration overhead have not been abandoned. In a nutshell, the fault tolerance and load balancing services need to be implemented. The implementation needs to be integrated into the used middleware. The server application needs to be updated, so that it will be able to support object group. The client has to undergo slight changes.

To elaborate, the middleware and the application need to be modified to support load balancing. The modifications include the implementation of the Load Balancing Service and integrating the service into the existing middleware infrastructure. The server-side application, the main CORBA services (mainly, the naming service and the transaction Service), and the client-side needs to be updated. The binding mechanism needs to be modified to support the introduction of the object groups. The server application, which initially binds instances of server implementation to the naming service, has to be changed. Instead, the client's requests need to be bound to the replica the load balancer selects. Hence, this requires modifications to the standard CORBA services through introducing protocols and interface that abides to the OMG standards. In an environment where several hosts are used to store the server objects, different object groups need to be created. The server application needs to be modified to populate servant instances. Additional interfaces need to be introduced in the IDL (Tables B-1 and B-2). ORB interceptors and initializers have to be implemented. On the client side, the client application needs to be modified to look up the load balancer instead of the naming service to get a replica object

reference. The load balancer will be then able to send an object reference by using the CORBA ForwardRequest exception that the client can catch. To configure, all the instances of JacORB over the different hosts have to be shutdown. To compile and package the developed services, an Ant script has to be updated for each service. This introduces additional 200 lines of code. The properties file (i.e., jacorb.properties) has to be updated on each host. These updates concern the ORBInitRef property and the interceptors ORBInitializer. All the JacORB instances then need to be restarted. Interested reader may refer to the appendix B for further details.

Table 6.9. Scalability in the CORBA-induced architecture: aggregate results

Task	SLOC
Fault Tolerant implementation	5117
Load Balancing implementation	3943
Server-side application (Server objects Implementation and Server application- on each host)	170
Client-side application	30
Configuration on each host	Stop/restart, 200 SLOC+ 13/host

6.3.4.2 Scaling the J2EE-Induced Architecture

In subsequent sections, we investigate how scalability could be achieved in the J2EE-induced version through replication mechanisms. We analyze the impact of the scalability change on the J2EE-induced architecture of the Duke's Bank.

Scalability in J2EE through replication

Figure 6.7 depicts a common J2EE [Sun Microsystems Inc., 2002] cluster architecture. Clustering enables a group of (typically loosely coupled) servers to operate logically as a single server. The advantages of clustering include the elimination of a single point of failure; the high service availability if multiple servers in the cluster can handle the same service; and load balancing by diverting requests to the least loaded server hosting the same service. We use JBoss 3.0[http://www.jboss.org/], an open source J2EE application server. JBoss clustering aims at improving scalability and high availability using replication techniques. JBoss relies on Jgroups

[http://www.jgroups.org/] for the clustering of its naming registry face- Java Naming and Directory Inter (JNDI)-and its EJB container. JGroups is an open source group communication middleware fully written in Java. JGroups provides the following main features: group creation and deletion, where group members can be spread across LANs or WANs; joining and leaving of groups; membership detection and notification including joined/left/crashed members; detection and removal of crashed members; sending and receiving of member-to-group messages (point-to-multipoint); and sending and receiving of member-to-member messages (point-to-point).

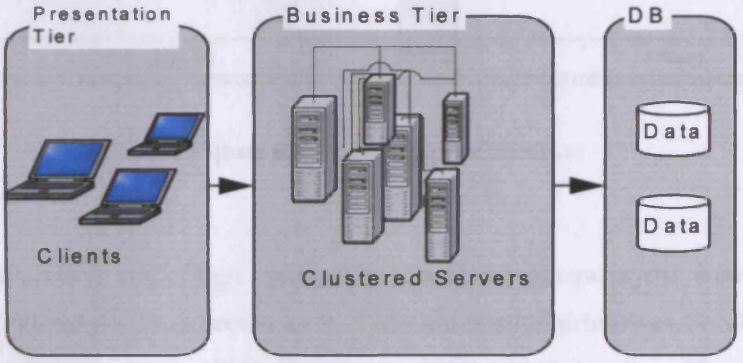


Figure 6.7. Example of J2EE cluster architecture

JBoss uses a layered architecture to manage clustering. The architecture relies on JGroups for clustering, which is abstracted. Figure 6.8 describes the architecture using two nodes. The term partition is used to refer to a cluster. A node can be part of several partitions.

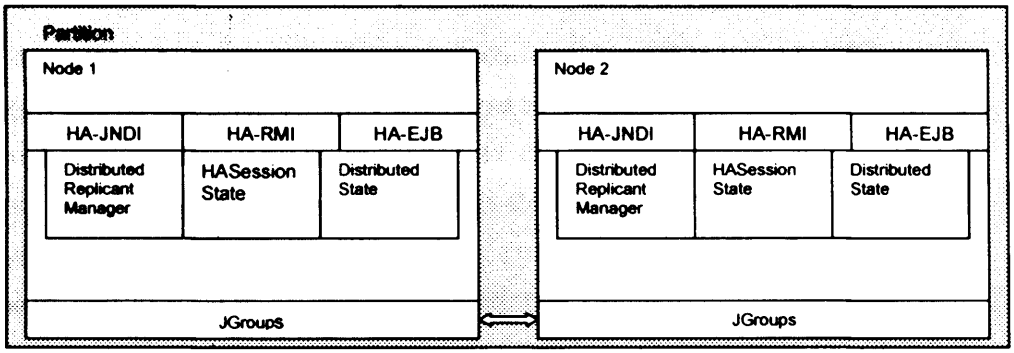


Figure 6.8. Clustering Architecture

The HAPartition (i.e., High Availability Partition) abstracts the communication framework; it provides access to a set of communication primitives. Services need to register with the HAPartition to use the HAPartition services. The Distributed Replicant Manager manages the replicas by providing methods to add or remove replicas from a partition. The HASession-State is used to manage the state of Stateful Session Beans. The state of all Stateful Session Beans are replicated and synchronised across the cluster each time the state of a bean changes. The Distributed State stores settings or parameters that should be used by the containers in the cluster. Clients can use either the local JNDI service or the HA-JNDI service to look up objects. If the local JNDI service is used, the local JNDI namespace is used to locate objects. HA-JNDI delegates the lookup to the local JNDI, if it fails to find the object within global the cluster-wide context. EJB homes are bound to the local JNDI of the server on which the particular EJB is deployed. HA-RMI provides load-balancing and fail-over facilities for RMI servers. HA-EJB allows selecting the load-balancing policy to apply (e.g., Round Robin, First Available), when deciding on a replica that will respond to the client request. The load-balancing policy is not adaptive. JBoss provides clustering for the two main types of EJB: Entity Bean and Session Bean (Stateful and Stateless). Clustering for Message-Driven Bean is not

provided yet. Also, JBoss comes with a farming feature. Farming manages cluster hot-deployment. Hot-deploying an application (EAR, WAR or JAR application) on a machine causes the application to be hot deployed on all instances within the cluster.

Change impact analysis

An observable advantage of scaling the software architecture induced by J2EE, using JBoss, is that no development effort is required to realize the scalability requirements through replication, as when compared to the CORBA version. The clustered environment, which mainly includes the HA-JNDI, the HA-EJB for Entity Bean and Stateful Session Bean, and the farming do provide the primitives for scaling the software system. That is, no development effort is required to provide a clustering environment. However, configuring and deploying the application in the clustered environment are still required.

In brief, configuration includes the following: configuring clusters, HA-JNDI, HA-EJB, and farming. By default, one partition exists. When adding a partition, the cluster needs to be configured. This simply requires updating the cluster ser-vice file (i.e., cluster-service.xml). Eleven lines of code are necessary to map a partition with a HA-JNDI service. The property file (jndi.properties) on the client-side has to be updated to enable the client to auto-discover the HA-JNDI servers. One line of code is necessary to update this file.

To cluster the EJBs, a special XML tag (clustered) has to be added to the Jboss.xml. To specify the partition(s) to be used, the (clusterconfig) tag needs to be added to the same file. More, the load-balancing mechanism may need to be up-dated in the JBoss deployment descriptor. All of these changes involve 10 lines/bean. For stateful session beans, the cluster service file, cluster-service.xml, need to be updated to add a partition to the HASessionState service, involving 7 SLOC. Therefore, we need 39 SLOC to enable farming for all our partitions. The file farm-service.xml file, by default, enables the farming for one partition. To enable the farming for all the partitions, farm-service.xml file need to be updated; a link will need to be added between the FarmMemberService and a partition. For the Duke's Bank architecture, we use four partitions: two for the Account beans (Entity and Session) and two for

the Transactions beans (Entity and Session). Thirty-two SLOC need to be added for configuring a partition. This results in 128 SLOC. Other 33 lines of code are necessary to map a partition with the HA-JNDI service. Because four kinds of beans exist in the system, configuring the HA-EJB requires 40 lines to update the JBoss deployment descriptor of the beans. Thirteen SLOC are required. We note that Farming is not enabled by default, requiring the developer's intervention. Table 6.10 aggregates the above description.

Table 6.10. Scalability in the J2EE version

Changes to make	4 partitions Source Lines of code (SLOC)
Install Jboss	1
Configuring clusters	96
Configuring HA-JNDI	34
Configuring HA-EJB	47
Configuring farming	39
Total for one host	217

6.3.5 The Throughput Valuation Point of View

A possible way to treat scalability is to assume that scalability can be measured by *throughput* or capacity of the system. Throughput is a generic performance criterion, which expresses the amount of work performed by the system under test during a unit of time. This criterion is based on the observation that for a fixed system with a given throughput (e.g., a single host), there is an inverse relationship between the response time and the number of clients. In other words, the more clients submitting requests, the longer are the delays.

A well-known throughput metric is the Total Operations per Second (TOPS) completed during the measurement interval, referred to as TOPS [http://www.spec.org/]. TOPS is composed of the total number of business transactions completed in the customer domain, added to the total number of work

orders completed in the manufacturing domain, normalized per second[http://www.spec.org/].

To understand how Duke's architecture may behave once induced with J2EE or CORBA, we have screened relevant performance benchmarks (e.g., Denaro et al., 2004; http://www.spec.org/jAppServer2005/; Shipping et al., 2005). We appeal to the use of published benchmarks, because the system of the given architecture need not be implemented during the evaluation. Thus, performance measures may not be available. Benchmarks are revealing on the performance dimension because, for example, if multiple benchmarks are conducted with a suitable mix of relevant factors, it may be possible to obtain a set of basic scalability results that can be used for estimating the throughput of possible configurations of the architecture. Depending on the benchmarking algorithm, the relevant scalability factors can be, for example, the number of objects, the number of clients, or the number of nodes in the system etc. supported in response to growing load. A major problem in comparing benchmark results, however, is that different hardware platforms and configurations (e.g., memory, disk drives etc) often produce different results making the comparisons difficult. Further, vendors often try many different ways to optimize performance, including adding cache memory and putting cache buffers on disk arrays. Therefore, we only use benchmarks, which are close to the case at hand. We then normalize the screened benchmarks for easing the comparison. It could be also argued that in iterative development (e.g., in the Unified Process) partial implementations might be available at the end of each phase. In this context, it is possible to create benchmarks from the partial implementations and to use them to recalibrate the screened ones. The intention is to have more meaningful figures which we could use for understanding the impact of likely change in future load on the behavior (throughput) of the system(i.e., relevant to the *throughput valuation point of view).*

In the context of ArchOptions, our use of benchmarks resembles the use of a twin asset. We argue that using benchmarks satisfies the concept of the twin asset as we are relying on historical information showing possible variations in performance in connection to change in load and relative to the candidate implementations. These benchmarks often hint that the throughput is dependent on and can be estimated

from the middle-tier "processing power" of the architecture. Such variation, we believe, is a wealth as it reveals pros and cons of the Duke's Bank execution under possible operating environments and/or in relation to other participating applications. This is advantageous because scalability is also a factor of the number of independently developed applications that might share an execution platform. The advantage of this approach is that the published benchmarks could reveal risks of the operating environment on the choice.

Figure 6.9 shows the likely throughput trend that the J2EE-induced architecture may exhibit relative to the CORBA-induced one, upon varying the TOPS and the number of hosts. For the J2EE-induced architecture, we provide throughput estimations for two possible implementations: one with JBoss and the other with WLS. For the CORBA-induced architecture, we provide estimates upon the use of JacORB to induce the architecture. Table 6.11 depicts the upper limit of TOPS supported per host for each of WLS, JBOSS, JacORB induced architectures for 1 to 4 hosts.

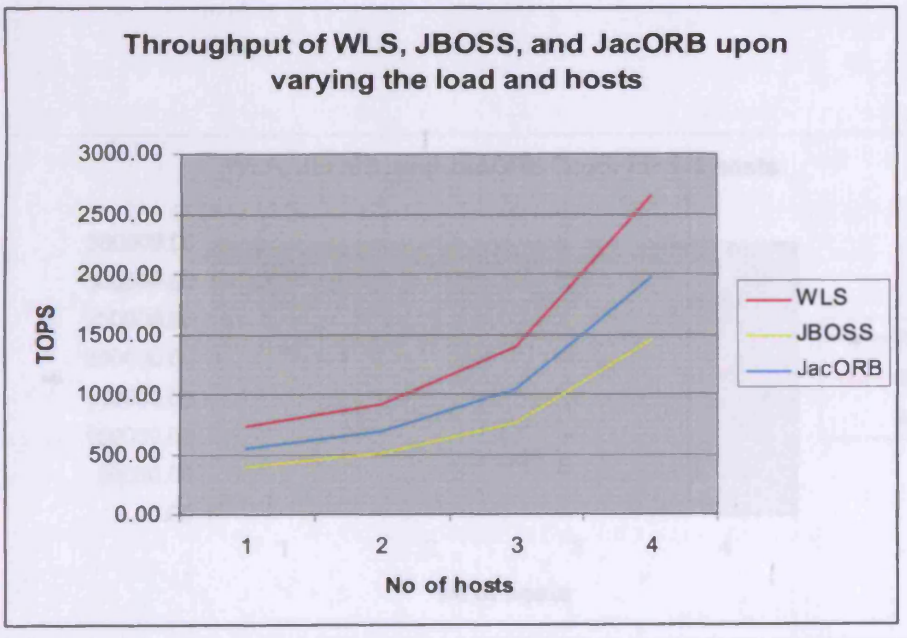


Figure 6.9. Plotting the TOPS per host for each of WLS, JBOSS, JacORB for 1 to 4 hosts

Table 6.11. Upper limit of TOPS per host for each of WLS, JBOSS, JacORB

Hosts	WLS	JBOSS	JacORB
1	732.00	400.26	546.80
2	918.36	502.16	686.01
3	1395.44	763.03	1042.39
4	2640.96	1444.08	1972.79

Figure 6.10 shows the likely cost-trend upon inducing the Duke's bank architecture with J2EE (using either WLS or JBOSS) and with CORBA (using JacORB). The likely cost is plotted against the number of hosts (1 to 4). The cost refers to the lifecycle cost of the System Under Test (SUT). The cost includes Application Servers/Containers, Database Servers, network connections, etc. Assuming, for example, a five-year lifecycle, cost would include all hardware (purchase price), software including license charges, and hardware maintenance. For the CORBA version, it assumed that the investment incurs an upfront cost to the development of the replication mechanism to support fault-tolerance and load-balancing services for high load scenarios. For the J2EE version of WLS, a license cost is incurred per host.

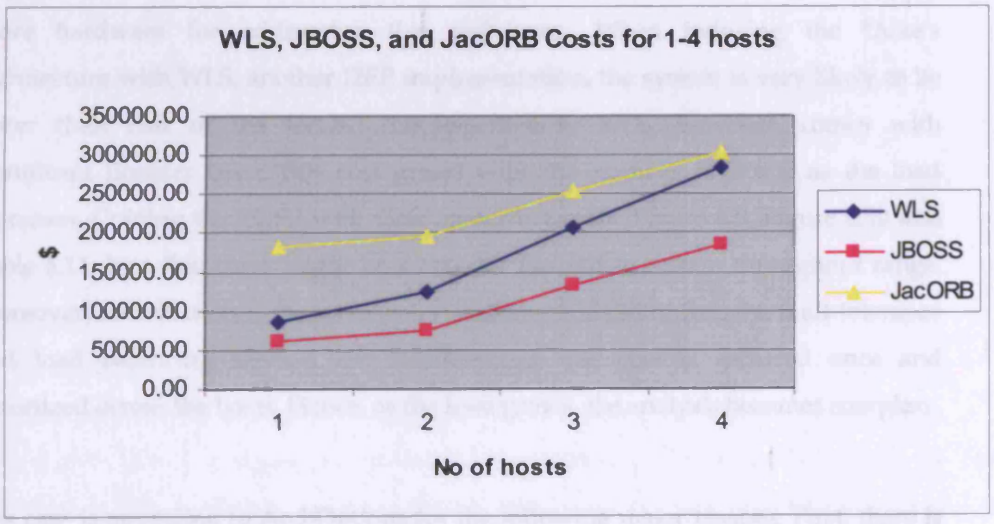


Figure 6.10. The likely cost-trend upon inducing the Duke's bank architecture with J2EE-(WLS or JBOSS) and with CORBA (JacORB).

6.3.6 Applying ArchOptions

In previous sections, we have seen that to scale the architecture of the Duke's Bank, the requirements depicted in Figure 6.4 need to be maintained. We have estimated their structural impact on both the CORBA and the J2EE versions. We have estimated the SLOC to be added for implementing the change on both versions, as depicted in Tables B-1, B-2, Table 6.9, and Table 6.10. From the structural valuation point of view, an observable advantage of scaling the software architecture induced by EJB is that no development effort is required to realize the scalability requirements through replication, as when compared to the CORBA version. J2EE provides the primitives for scaling the software system, which result in making the architecture of the software system more *flexible* in accommodating the change in scalability requirements, as when compared to the CORBA version. Though the structural analysis appears to be in favor of the J2EE-induced architecture, the throughput analysis may reveal a different trend. From the throughput valuation point of view, Figure 6.9 shows that when the Duke's architecture will be induced with JBoss, a J2EE implementation, the system is likely to be slower than that of the JacORB one. This is because JBOSS uses reflection [http://www.jboss.org]. This also implies that there are some chances for the JBoss-induced architecture to require more hardware for addressing this deficiency. When inducing the Duke's architecture with WLS, another J2EE implementation, the system is very likely to be faster than that of the JacORB implementation. WLS, however, comes with significant licenses costs; this cost grows with the number of hosts, as the load increases. Coining the TOPS with their associated costs, Figure 6.9, Figure 6.10 and Table 6.11, hint that there might be a case for JacORB in certain throughput range. Moreover, note that once the services for realizing scalability (i.e., the fault-tolerance and load balancing service) are implemented, the cost is incurred once and amortized across the hosts. Hence, as the load grows, the analysis becomes complex.

The case is appealing to ArchOptions for the following major reasons: First, there is cone of uncertainty associated with the growing load and consequently in the value added as result of our choice. Moreover, the TOPS are of straightforward contribution to value. That is, the more operations are completed per second, the

more value is added to the enterprise. However, TOPS incur a price upon executing the operations. The price again is dependent on several factors such as the number of hosts, the hardware, the license cost, and any additional costs that are necessary for making the middleware adaptable to the growing load. In the context of the Duke's Bank, the TOPS range is often uncertain as it is dependent on the customers' behavior at a time. The *uncertainty* in the likely range (i.e., TOPS), the associated costs for executing the TOPS, and the "fluctuation" in the value added as a result make the case very appealing to the use of ArchOptions. For the throughput valuation point of view, the analysis using ArchOptions aims at complementing the behavioral analysis to understand the trend in the added value upon embarking on either J2EE (Jboss or WLS) or CORBA(JacORB) to induce the architecture of a given system.

6.3.6.1 Formulation and Interpretation

In this section, we describe how ArchOptions can be tailored to understand the value added as a result of inducing the architecture by EJB relative to CORBA, if the change in scalability requirements materializes and relative to the two valuation points of view.

As we have noted in [Bahsoon and Emmerich, 2003a; Bahsoon 2003; Bahsoon and Emmerich 2004a; Bahsoon and Emmerich 2004b], the search for a potentially stable architecture requires finding an architecture that maximizes the yield in the added value, relative to some future changes in requirements. As we are assuming that the added value is attributed to flexibility, the problem becomes maximizing the yield in the embedded or adapted flexibility in a software architecture relative to these changes. For this case study, given the choice of two or more middleware candidates, the selection has to maximize the yield in the embedded or adapted flexibility in response to likely changes in scalability requirements. In particular, a proper selection has to maximize the value added relative to the two valuation points of view. That is, the decision to select potentially stable-middleware architecture has to provide a compromise between the payoff on the structural and the behavioral valuation points of view, as we will see in the subsequent Sections.

Let us assume that we are given the choice of two middleware M_0 and M_1 to induce the architecture of a particular system. Let us assume that S_0 and S_1 are the architectures obtained from inducing M_0 and M_1 respectively. Say, inducing M_1 is an economical choice, if it adds value to S_1 relative to S_0. We attribute the added value to the enhanced flexibility of S_1 over S_0. If we are considering stability as a criteria for understanding the value added on the system, then future changes in non-functional requirements will tell us how valuable S_1 is relative to S_0, as we are performing a tradeoff between the architecture induced by M_0 and M_1. However, the added value is uncertain, as the demand and the nature of the future changes are uncertain. Hence, using option theory is a promising approach to inform the selection.

Choosing a particular middleware to induce the architecture of the software system can be seen as an investment to purchase flexibility in the induced software architecture. The non-functional requirements and the range in which they change influence the choice. In this context, deciding on a particular middleware to induce the software system architecture can be seen as an investment to purchase future *growth options* that enhance the upside potentials of the structure when the non-functional requirements change. That is, S_1 is said to be more accommodating to the change than S_0, if S_1 holds more growth options than S_0. For a valuation point of view *p*, we focus the analysis on the calls of the ArchOptions model for valuing the growth options, as given in *(6.2)*.

$$\sum_{i=1\ldots n} E\,[\max\,(x_i V_p - C_{eip},\, 0)] \qquad (6.2)$$

The selection has to be guided by the expected payoff in $(\sum_{i=1\ldots n} E\,[\max\,(x_i V_p - C_{eip},\, 0])_{S1}$ relative to that of $(\sum_{i=1\ldots n} E\,[\max\,(x_i V_p - C_{eip},\, 0])_{S0}$. That is, if $(-I_e + \sum_{i=1\ldots n} E\,[\max\,(x_i Vp - C_{eip},\, 0)]_{S1} > \sum_{i=1\ldots n} E\,[\max\,(x_i V_p - C_{eip},\, 0)]_{S0})$ for some likely changes, then it is worth investing in M_1, as the investment in M_1 is likely to generate more growth options for S_1 than for S_0 and relative to the *p* valuation point of view.

If (E [max ($x_k V_p$ - C_{epk}, 0)])$_{S1}$=0), then M_1 is not likely to payoff, relative to M_0, as the flexibility of the architecture to the change is not likely to add a value for S_1 on *p*, if the change need to be exercised. Two interpretations might be possible: (i) the architecture is overly flexible in the sense that its response to the change(s) has not "pulled" the options relative to *p*. This implies that the embedded flexibility (or the resources invested in implementing flexibility- if any) are wasted and unutilized to reveal the options relative to the changes and relative to *p* (ii) the other case is when the architecture is inflexible relative to the change. This is when the cost of accommodating the change on S_1 is much more than the cumulative expected value of the architecture responsiveness to the change.

For the maintainability valuation point of view, *PM*, we appeal to the use of future savings in maintenance effort as a way to quantify the value added due to a selection. If we assume that $x_i V_{PMS1}$ is the expected savings in S_1 over S_0 due to selection, then if ($\sum_{i=1...n}$ E [max ($x_i V_{PM}$- C_{eiPM}, 0)] $_{S1}$ > $\sum_{i=1...n}$ E [max ($x_i V_{PM}$- C_{eiPM}, 0)] $_{S0}$), then investing in M_1 is said have better value with respect to PM. For the throughput valuation point of view, *Pthro*, an additional operation is said to "buy" an architectural potential paying an exercise price. In terms of throughput, the architectural potential is a performance measure. That is, the more TOPS are said to be completed at a host (or for a configuration), the more value is said to be added to the enterprise. The more valuable is said the architectural potential relative to the TOPS. The exercise price is price/TOPS (see relevant section for more details). If we assume that $x_i V_{PthroS1}$ is the value added in S_1 over S_0 due to the support of more TOPS, it is reasonable to consider that if ($\sum_{i=1...n}$ E [max ($x_i V_{Pthro}$ - $C_{eiPthro}$, 0)] $_{S1}$ > $\sum_{i=1...n}$ E [max ($x_i V_{Pthro}$ - $C_{eiPthro}$, 0)] $_{S0}$), then investing in M_1 is said to payoff relative to throughput valuation point of view.

6.3.6.2 Options on the Maintainability Valuation Point of View

For this valuation point of view, we aim at understanding the value added upon inducing the architecture with EJB relative to CORBA, if the change in scalability requirements materializes. We use future savings in maintenance, deployment, and configuration costs (if any), upon accommodating the likely change in scalability, as a way to quantify the value added. Below, we show how we estimate the parameters relative to this valuation point of view.

Upon applying ArchOptions, we focus our attention on the payoff of the call options (i.e., $\sum_{i=1...n} E\,[\max(x_i V_{PM} - C_{eiPM}, 0)]_{S1}$ relative to $\sum_{i=1...n} E\,[\max(x_i V_{PM} - C_{eiPM}, 0)]_{S0}$), as they are revealing for the flexibility of the architecture-induced in responding to the likely future changes. We construct a call option for the future scalability goal, where the change is analogues to buying an "architectural potential", paying an exercise price. The exercise price corresponds to the likely price to accommodate the change in load on the structure. When necessary, we use $6000 for man-month to cast the effort into cost. We show how we have estimated the parameters.

Table 6.12. Scaling the system using replication (1 Host): development, configuration, and deployment costs

		CORBA (JacORB)			EJB (JBOSS)		
		Optimistic	Most Likely	Pessimistic	Optimistic	Most Likely	Pessimistic
Development	Effort	24.1	30.2	37.7	0	0	0
	Cost, C_{eiPM}	96481	120602	150753	0	0	0
	SLOC	9240			0		
Configuration & Deployment	Effort	0.4	0.5	0.6	0.4	0.5	0.6
	Cost, C_{eiPM}	1527	1909	2386	1558	1948	2435
	SLOC	213			217		

Estimating (C_{eiPM}). The exercise price corresponds to the cost of implementing scalability on each structure, given by C_{eiPM} for requirement *i*. As the replicas may

need to be run on different hosts, we devise a general model for calculating C_e as a function of the number of hosts, given by:

$$C_{eiPM} = \sum_{h=1...k} (C_{dev}, C_{config}, C_{deploy}, C_{licesh})_h \qquad (6.3)$$

where, h corresponds to the number of hosts. C_{dev}, C_{config}, *and* C_{deploy}, respectively corresponds to the cost of development (if any), configuration, and deployment for the replica on host h. C_{licesh} corresponds to licenses and hardware costs, if any. All costs are given in ($). We provide three values: optimistic, likely, and pessimistic for each parameter. All are calculated using COCOMO II - post architectural model [Boehm et al., 1995], as depicted in Table 6.12. Upon varying the number of hosts, we only report on pessimistic values for this study, as they are revealing.

Estimating (x_iV_{PM}). To value the architectural potential of S_1 relative to S_0 given by ($x_iV_{PMS1/S0}$), we take a structural approach to valuation. We use the expected savings (if-any) in development, configuration, and deployment efforts, when the scalability change needs to be accommodated on S_1 relative to S_0, and respectively denoted as $\Delta_{S1/S0}C_{dev}$, $\Delta_{S1/S0}C_{config}$, $\Delta_{S1/S0}C_{deploy}$. Relative savings in licenses may also be considered and denoted by ΔC_{licesh}. Below is a model for calculating $x_iV_{S1/S0}$, for the change in requirement i.

$$x_iV_{PM\,S1/S0} = \sum_{h=1...k} (\Delta_{S1/S0}C_{dev}, \Delta_{S1/S0}C_{config}, \Delta_{S1/S0}C_{deploy}, \Delta_{S1/S0}\,C_{licesh})_h \qquad (6.4)$$

Similar description applies for ($x_iV_{PMS0/S1}$). The savings (if any), however, are uncertain and differ with the number of hosts, as the replicas may need to be run on different hosts. Such uncertainty makes it even more appealing to use of "options thinking".

Estimating volatility (σ_{PM}). The volatility of the stock price is a statistical measure of the stock price fluctuation over a specific period of time; it is a measure of how

uncertain we are about the future of the stock price movements. Volatility stands for the fluctuation in the value of the estimated $x_i V_{PM}$. Intuitively, it aggregates the "potential" values of the structure in response to the change(s). We adhere to the real options principles in estimating σ_{PM}. We take the percentage of the standard deviation of the $x_i V_{PM}$s estimates-the optimistic, likely, and pessimistic values to calculate σ_{PM}.

Exercise time (t_{PM}) and free risk interest rate(r_{PM}). As a simulation assumption, we set the exercise time to one year, assuming that the Duke's Bank needs to accommodate the change in one year time. We set the free risk interest rate to zero (i.e., assuming that the value of money today is the same as that in one year's time).

6.3.6.3 Options on the Throughput Valuation Point of View

We take throughput as a measure for analyzing the payoff on the behavioral point of view. We construct call options for a likely change in load-range. The objective is to analyze the architectural potential in supporting a likely growth of TOPS. Below, we show how we estimate the parameters relative to this valuation point of view.

Estimating ($C_{eiPthro}$). A change in a load-range is said to buy an architectural potential paying an exercise price. As we mentioned before, TOPS denotes the Total Operations completed per Second. For the simplicity of explanation, let us assume that the system of the induced architecture needs to scale up to support an additional operation per unit-time. An additional operation is said to buy an architectural potential paying an exercise price. In terms of throughput, the architectural potential is a performance measure. Hence, what an extra operation pays, if materializes, is a bandwidth for performing that operation. Inducing the Duke's bank with either J2EE or CORBA provide different bandwidth capabilities for performing the operation at different price. If the implementation of either happens to hold embedded growth options in supporting the extra operation, then the operation is said to pay an exercise price to buy options on the architecture. To estimate the exercise price, we use a well-known normalization factor, which is the *price/performance* [http://www.spec.org/jAppServer2005/] (i.e., the lifecycle cost of the System Under

Test (SUT) as configured for the benchmark divided by the throughput). As an example, assuming five-year lifecycle, the cost would include all hardware (purchase price), software including license charges, and hardware/software maintenance. If the total price is $5,734,417 and the reported throughput is 105.12 TOPS, then the price/performance is $54,551.16/TOPS (54,551.151 rounded up).

Estimating ($x_i V_{Pthro}$). For simplicity, we estimate $x_i V_{Pthro}$ relevant to the business domain. For every completed on-line operation, Duke's would not need to have to serve a customer in person at a branch. That is, the Duke's savings are in the manual-effort for serving the clients at a branch. For example, let us assume a scenario where a clerk needs one minute for completing a business operation: if we assume an overhead cost of $100,000/year for each clerk, then an online operation saves about a dollar per operation in a minute: $100000/ (220day * 8hours * 60minutes). Computing the savings per second is then straightforward. We use scenarios of 8 and 20 clerks for computing $x_i V_{Pthro}$.

Estimating volatility (σ_{Pthro}). Volatility represents uncertainty attributed to the likely growing of load. For some computation, we abide to the real options principles in computing volatility: we use the standard deviation of $x_i V_{Pthro}$s due supporting extra operations for a range of load at a particular host (as the range is said to be revealing to the fluctuation in the value). For some computations, we use modeling estimates for volatility, representing uncertainty, with the objective of demonstrating how volatility is said to influence the options results.

Exercise time (t_{Pthro}) and free risk interest rate(r_{Pthro}). As a simulation assumption, we set the exercise time to one year, assuming that the Duke's Bank needs to accommodate the change in one-year time. We set the free risk interest rate to zero (i.e., assuming that the value of money today is the same as that in one year's time).

6.3.7 Options Analysis: Results and Discussion

In this Section, we report on some selective results and observations upon the application of the model. As part of this evaluation, the objective of this section is to

extend the confidence in some of the claims that ArchOptions makes and to simulate the application of the model. These claims are sufficiently described in Section 6.1 of this Chapter. In particular, We verify that the choice of a stable distributed software architecture has to be guided by the choice of the underlying middleware and its *flexibility* in responding to future changes in non-functional requirements. We verify the hypothesis that flexibility creates real options in the structure relative to likely changes in requirements. We exemplify the use of valuation point of view for capturing the options from different perspectives. We demonstrate how uncertainty impacts value and consequently the decision of selecting a stable architecture. We show how the options results are compared to other valuation techniques, which fall short in dealing with the value of flexibility under uncertainty. In line of previous discussion, CORBA and J2EE correspond to M_0 and M_1 respectively. We refer to the architecture of the Duke's Bank as S_0 when induced by M_0 and S_1 when induced M_1.

Observation 1. Flexibility creates real options: S_1 is more flexible than S_0 (due to the primitives in J2EE); S_1 has created more real options than S_0.

Let us first focus the analysis on the *maintainability valuation point of view, PM.* Let us consider the scenario where we consider one host. For this scenario, we assume that the license cost (C_{licesh}) is zero for M_1 (e.g., the usage of JBoss an open source). Table 6.12 reports on the effort (man-month) and cost in ($); it provides three values: optimistic, likely, and pessimistic for each parameter. The $x_i V_{PMS1/S0}$ correspond to the difference- as reported in Table 13a. The overall expected savings of inducing the structure with S_1 relative to S_0 are in the range of $96450(pessimistic) to $150704(optimistic). As far as the development effort is concerned, expected savings are in the range of $96481(pessimistic) to $150753(optimistic) for realizing the scalability requirements. As far as configuration effort is concerned, S_1 has not reported any expected savings relative to S_0. However, these figures are insignificant. As far as the effort of deployment is concerned, both are comparable when it comes to SLOC. We note that these figures are based on COCOMO II: the number of man-months is different from the time that will take for completing a project, termed as the development schedule. For example, a

project could be estimated to require 50 man-months of effort but have a schedule of 11 months. Accordingly, the cost and relative savings, maybe adjusted relative to the schedule. We have relaxed this, as the aim of the exercise is to simulate the applicability of the model. The $x_i Vs$ will be used to quantify the added value, taking the form of options, due to the embedded flexibility on S_1 relative to S_0.

Table 6.13a shows that S_1 is in the money in response to the change in scalability, when compared to S_0. Table 6.11a shows that S_1 is in the money relative to the development, configuration, and the deployment. The results of table 6.13a read that inducing the architecture with M_1 is likely to enhance the option value by an excess of $96450(pessimistic) to $150704(optimistic) over S_0, if the change in scalability need to be exercised in one year time. Thus, the results show that S_1 induced by M_1 is likely to add more value in the form of options in response to the change, when compared to S_0. It is worth pointing out that though S_1 is flexible relative to the scalability change, it might not necessarily mean that it might be flexible with respect to other changes. Obviously, JBoss does provide the primitives for scaling the software system, which result in making the architecture of the software system more flexible in accommodating the change in scalability, as when compared to the CORBA version. This has lead to a notable savings in maintenance cost. Calculating the options of S_0 relative to S_1, we can see that S_0 is said to be out of the money for this change. The CORBA version has not added value, relative to J2EE, as the cost of implementing the change was relatively significant to "pull" the options, as reported in Table 6.13b. The very low value of Vega means that possible changes in volatility have relatively little impact on the value of the options. The high value of Delta in Tables 13a and Table 6.13b roughly means that changes in $x_i V_{PM}$ could have high impact on the on the calculated options.

Table 6.13a. The options in ($) on the architecture induced by S_1 relative to S_0 for one host, with S_1 license cost (C_{licesh}) =zero for the maintainability valuation point of view

		C	V	σ	I	Options	Delta	Vega
Overall	Optimistic	1158	96450	22.7	1	94892	1	9.1149E-71
	Likely	1948	120563			118615	1	1.1628E-70
	Pessimistic	2435	150704			148269	1	1.4533E-70
Development	Optimistic	0	96481	22.7	1	96481	1	0
	Likely	0	120602			120602	1	0
	Pessimistic	0	150753			150753	1	0
Configuration and Deployment	Optimistic	1558	-31	22.7	1	0	0	0
	Likely	1948	-39			0	0	0
	Pessimistic	2435	-49			0	0	0

Table 6.13b. The options in ($) on the architecture induced by S_0 relative to S_1 for one host, with (C_{licesh}) =zero for the maintainability valuation point of view

		C	V	σ	I	Options	Delta	Vega
Overall	Optimistic	96450	31	22.7	1	0	0	0
	Likely	120563	39			0	0	0
	Pessimistic	150704	49			0	0	0

Table 6.13c. Options in ($) on S_0 relative to S_1 with (C_{licesh}) = $25000 and σ_{PM}=22.7 *and pessimistic* C_{eiPM} for the maintainability valuation point of view

	C		Options	Adjusted Options	Concurrent Users
1	2386	25049	2343	0	$U1S_0$ vs $U1S_1$
2	4772	50049	4772	0	$U2S_0$ vs $U2S_1$
3	7158	75049	67891	0	$U3S_0$ vs $U3S_1$
4	9544	100049	90505	0	$U4S_0$ vs $U4S_1$
5	11930	125049	113119	0	$U5S_0$ vs $U5S_1$
6	14316	150049	135733	0	$U6S_0$ vs $U6S_1$
7	16702	175049	158347	7643	$U7S_0$ vs $U7S_1$

Let now us inspect another form of flexibility that S_1 provides over S_0, relative to the *throughput valuation point of view, Pthro*:

Consider a scenario, where the likely load is 1042 TOPS. Table 6.14a shows that 1042 TOPS can be supported by three hosts, if the Duke's architecture is induced with either M_1 (WLS) or M_0 (JacORB). Table 6.14a shows that for three hosts, supporting 1042 TOPS costs $1488.88 for S_1 when induced with WLS but $243.05 for S_0 when induced with JacORB. The cost is denoted by

$C_{eiPthro}$. Supporting 1042 TOPS online is assumed to eliminate manual-overhead and create x_iVs, as explained in Section 6.3.6 and computed using eight clerks scenario. Using high volatility modeling assumptions for σ_{Pthro}= 100% for simplicity, Table 6.14a shows that S_1 adds more value than S_0 for three hosts. This is because the cost of implementing both load balancing and fault-tolerance is far from breaking even on S_0 for three hosts.

Let us now suppose that Duke's can only afford to invest in three hosts and the investment is to be made. Let us now assume that the load is likely to grow from 1042 TOPS to the range of 1250-1395 TOPS, as a result of accommodating more customers in one year time:

According to Table 6.14b, as the load increases over 1042 TOPS, M_1 continues to be of a better value for flexibility as when compared to M_0 for the following reasons: First, S_0 will be inflexible to support an extra operation beyond 1042 TOPS for three hosts (Table 6.11). That is, the growing load requires an additional host; henceforth, incurring hardware costs. Second, the cost of implementing both load balancing and fault-tolerance is far from breaking even on S_0 for three hosts. As a result, S_0 ceases to create real options on three hosts if the load exceeds the expected 1024 TOPS. Conversely, for the range of 1250-1395 TOPS, S_1 tends to carry growth options on three hosts. This is because at threshold, S_1 can support around 1395 TOPS (Table 6.11). That is, S_1 when induced with WLS, tends to create value for an additional 371 TOPS on three hosts.

Formalizing this thinking,

> The architectural potential of S_1 (WLS) = value in supporting 1042 TOPS now + growth options in supporting an additional 371 TOPS;
>
> The architectural potential of S_0 (JacORB) = value in supporting 1042 TOPS now + zero growth options beyond 1042 TOPS.

Table 6.14a. Supporting 1042 TOPS with three hosts and their options value, if the Duke's architecture is induced with either M_1 (WLS) or M_0 (JacORB), σ_{Pthro}= 100%

1042 TOPS	No Hosts	C	V	Options
S_1(WLS)	3	148.88	131.61	45.44
S_1(JBOSS)	4	126.96	131.61	51.86
S_0(JacORB)	3	243.05	131.61	27.59

Hence, for three hosts and with the likely growing load in the range of 1250-1390 TOPS, S_1 exhibits that it has flexibility under uncertainty. This flexibility takes the form of growth options held on S_1. The value of these options is in supporting an additional 371 TOPS. The more uncertain we are about the likely growth in load (i.e., beyond 1024 TOPS and in the range of 1250-1390 TOPS), the more valuable is the flexibility in S_1 relative to S_0.

Table 6.14b. Supporting 1395 TOPS with three hosts and their options value, if the Duke's architecture is induced with either M_1 (WLS) or M_0 (JacORB) σ_{Pthro}= 100%

1250-1395 TOPS	No Hosts	C	V	Options	Growth Options
S_1(WLS)	3	148.88	176.61	77.05	31.61
S_1(JBOSS)	4	126.96	176.1	85.79	33.93 for 4 hosts
S_0(JacORB)	3	243.05	131.61	27.59	0

Observation 2. How worth is the embedded flexibility in S_1 when induced with M_1, relative to that of S_0 when induced with M_0?

Consider the case where we use WLS as M_1 with an average upfront payable license cost C_{licsh}= $25000/host. As an upfront license fee is incurred, increasing the number of hosts may carry unnecessary expenditures that could be avoided, if we use M_0 instead. Let us first analyze the case from the structural point of view: M_0 does also incur costs upon scaling the software system through the development of both the load balancing and the fault tolerance services. Such a cost, however, maybe "diluted" as the number of hosts increases. The cost is said to be distributed across the hosts and incurred once, as the developed services can be reused across other hosts. For

this experiment, we assume that developing the fault tolerance and load services are upfront investments to buy growth options on the structure. An additional configuration and deployment cost materializes per host and sum up to the exercise price, C_{eiPM} as in equation (6.3), when an additional host is needed to scale the software. $x_i V_{PMS0/S1}$ is calculated based on equation (6.4). We calculate the options of S_0 relative to S_1. We adjust the options by subtracting the upfront expenditure of developing both services on M_0, as reported in Table 6.13c. The adjusted options reveal situations in which S_0 is likely to add value relative to S_1, when the upfront cost is considered. These results may provide us with insights on the cost effectiveness of implementing fault tolerance and load balancing support to scale the software system relative to S_1, where a licensing cost is incurred per host. Therefore, a question of interest is: when is it cost effective to use M_0 instead of M_1 relative to the structural point of view (maintainability)? In other words, when the flexibility of M_1 cease to create value relative to M_0. We assume that for any k hosts, S_0 and S_1 are said to support U_kS_0 and U_kS_1 concurrent users, respectively; where U_kS_0 could be different or equal to U_kS_1. For the non-adjusted options results of Table 6.13c shows that inducing the architecture with M_0 is likely to enhance the option value of S_0 relative to S_1 (pessimistic) for the case of n hosts for n>0, under the condition that U_nS_0 >>=U_nS_1 and under the assumption that the upfront cost of developing fault tolerance and load balancing is relaxed. However, if we benchmark these options values against the cost of developing the load balancing and fault tolerance services (i.e., the upfront cost), we can see that payoff following developing these services is far from breaking even for less than seven hosts, as depicted in Figure 6.11.

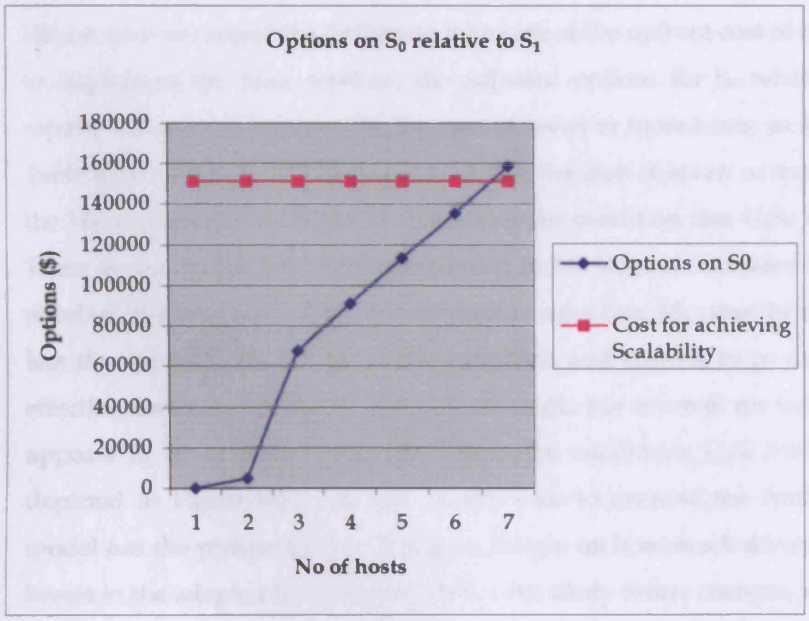


Figure 6.11. Maintainability valuation point of view: Options on S_0 relative S_1 prior to adjustment

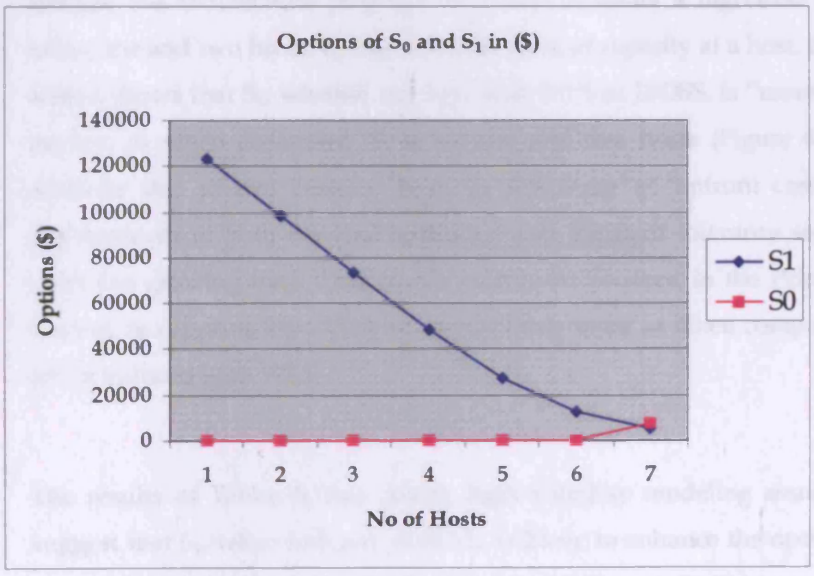


Figure 6.12. Maintainability valuation point of view: Options on S_0 and S_1 upon varying the number of hosts

Hence, once we adjust the options to take care of the upfront cost of investing to implement the both services, the adjusted options for S_0 relative to S_1 reports values in the money for the case of seven or more hosts, as shown in Table 6.13c and sketched in Figure 6.12. For the case of seven or more hosts, the M_0 appears to be a better choice under the condition that $U_nS_0 >>= U_nS_1$. These is due to the fact the expenditures in M_1 licenses increases with the number of hosts, henceforth, the savings in adopting M_1 cease to exist. For less than seven hosts, M_1 has better potentials and appears to be more cost-effective under the condition that $U_nS_1 >>= U_nS_0$. For seven or more hosts, M_0 appears to be of better potentials under the conditions $U_nS_0 >>= U_nS_1$, as depicted in Figure 6.12. The use of this case to exercise the ArchOptions model has the prospect in providing an insight on how much do we need to invest in the adapted flexibility relative to the likely future changes, while not sacrificing much of the resources.

Let us now turn to the throughput valuation point of view, PM: Let us analyze the architectural potential of S_1 and S_0 under a high-load scenario using one and two hosts. Under full utilization of capacity at a host, the value added shows that S_1, whether induced with WLS or JBOSS, is "more" in-the-money, as when compared to S_0 for one and two hosts (Figure 6.13). We attribute this to two reasons: First, S_0 will incur an upfront cost for the development of both the load balancing and the fault tolerance services to meet the growing load. This cost is said to be counted in the Price/TOPS. Second, S_0 supports less TOPS for one and two hosts, as when compared to S_1 when induced with WLS.

The results of Tables 6.15a-c (using high volatility modeling assumptions) suggest that S_1, when induced with M_1, is likely to enhance the option value by \$25.1751/second (when induced with WLS) and \$2.13/second (when induced with JBOSS) over S_0 for one host. The results also suggest that S_1 is likely to enhance the option value by \$30.2/second (when induced by WLS) and by \$1.4/second (when induced by JBOSS) over S_0 for two hosts, if the

change in load materializes in one year-time. The computation assumes a full utilization of capacity per host under a similar load. As the load is likely to grow, the results suggest that S_0 is likely to enhance the options value over S_1, when induced by JBoss by \$6.9/second and \$42.12/second respectively for three and four hosts. This is because under full utilization of capacity, S_0 is likely to support additional 279.36 TOPS using three hosts and another 528.71 TOPS using four hosts, as when compared to S_1, when induced with JBOSS. This implies that the adapted flexibility, due the development of the load-balancing and the fault-tolerant services on S_0, tend to be of better value than the "embedded" flexibility of S_1, when induced with JBoss. S_1, when induced with WLS, continues to be of a better value for three and four hosts as when compared to S_0. It enhances the value by \$48.3/second for three and by \$102/second for four hosts. The interpretation is as follows: First, WLS can support additional 353 TOPS on 3 hosts and another 668 TOPS on 4 hosts, as when compared to S_0. In terms of real options, WLS has embedded flexibility in supporting extra tops/hosts. That is WLS, has better value under uncertainty. Second, S_0 is less "performant" than S_1 (when induced with WLS); that is, S_0 can execute less TOPS and generate less value.

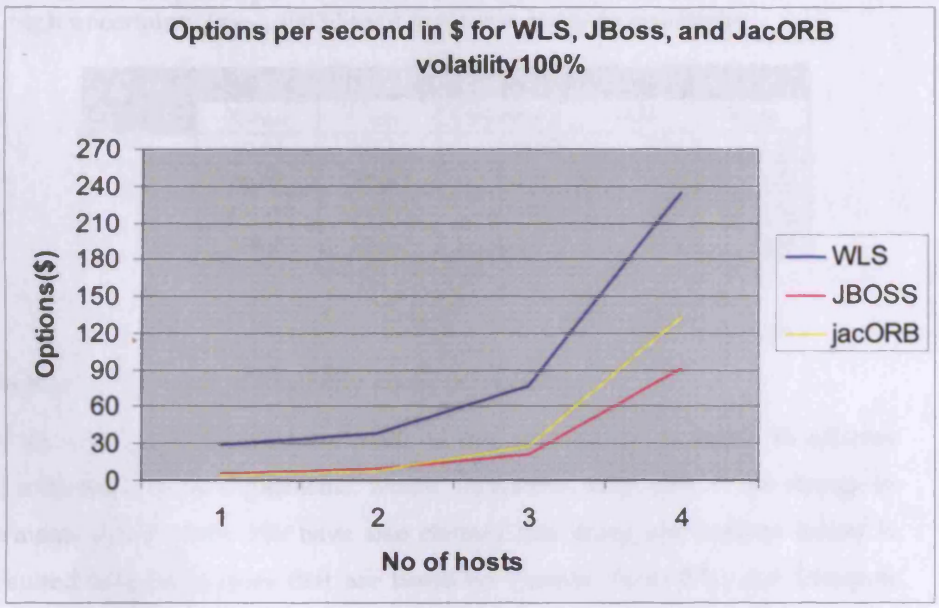


Figure 6.13. Throughput valuation point of view: Options per second ($) for WLS, JBoss, and JacORB under high volatility assumptions

Table 6.15a. Throughput valuation point of view: Options per second ($) for S_1 when induced with WLS under high uncertainty (σ_{Pthro} 100%) for 1 to 4 hosts and their sensitivity

Hosts	S_1 induced with WLS				
	$X_{ivPthro}$	$C_{eiPthro}$	$Options_{Pthro}$	Delta	Vega
1	92.42424	116.5451	28.639	0.60	0.35
2	115.9549	136.2558	38.3265	0.63	0.43
3	176.1919	148.8778	75.937	0.79	0.56
4	333.455	107.2882	234.9709	0.94	0.35

Table 6.15b. Throughput valuation point of view: Options per second ($) for S_1 when induced with JBOSS under high uncertainty (σ_{Pthro} 100%) for 1 to 4 hosts and their sensitivity

Hosts	S_1 induced with JBOSS				
	$X_{ivPthro}$	$C_{eiPthro}$	$Options_{Pthro}$	Delta	Vega
1	50.53758	150.68	5.60400	0.28	0.17
2	63.40412	149.62	9.78300	0.39	0.24
3	96.34174	173.98	20.73000	0.46	0.39
4	182.3332	126.96	90.38285	0.81	0.51

Table 6.15c. Options per second ($) for S_0 when induced with JacORB under high uncertainty (σ_{Pthro} 100%) for 1 to 4 hosts and their sensitivity

Hosts	S_0 induced with JacORB				
	$X_{ivPthro}$	$C_{eiPthro}$	$Options_{Pthro}$	Delta	Vega
1	69.04	330.86	3.46398	0.14	0.16
2	86.62	285.32	8.24432	0.24	0.27
3	131.61	243.05	27.59433	0.45	0.52
4	249.09	154.07	132.54728	0.84	0.61

Observation 3. The value of flexibility under uncertainty

One of the earlier claims we have made is that real options is suited to address typical software evolution problems, where uncertainty attributed to the change in requirements is the norm. We have also claimed that using real options theory is better suited than techniques that are based on Present Value (PV) and Discount Cash Flow (DCF) as these techniques tend to systematically underestimate the value of flexibility under uncertainty. As we have mentioned in several occasions, in our case the likely change in load is the major source of uncertainty that the Duke's Bank faces. To address such uncertainty and provide better insights on value creation, we have appealed to the use of real options theory.

$$DCF = \frac{\text{Cash Flow Year 1}}{(1+r)^{1}} + \frac{\text{Cash Flow Year 2}}{(1+r)^{2}} \ldots + \frac{\text{Cash Flow Year n}}{(1+r)^{n}}$$

Figure 6.14. The Cash flow at Year i, represents cash flows in which the cash flows occur, and r is a per-period discount rate

Let us assume that the load is assumed to be in the range of 30- 50 TOPS. Based on the benchmarks, 30-50 TOPS could be easily addressed by one host using either M_0 (JacORB) or M_1 (JBOSS or WLS). Figure 6.15 sketches the likely associated costs when inducing the architecture with either alternative.

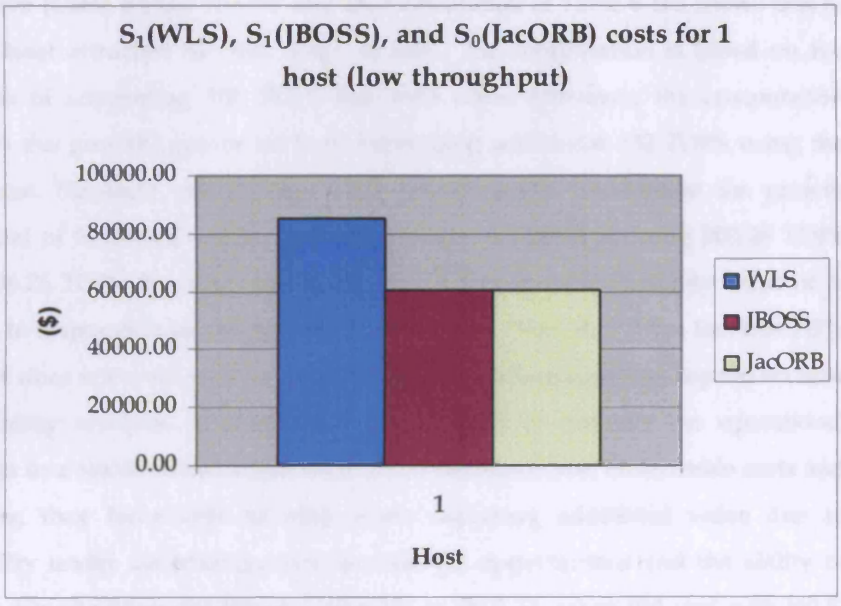


Figure 6.15. The likely associated costs compared upon inducing Duke's architecture with WLS, JBOSS, and JacORB for very low throughput requirements on 1 host

For such a low throughput requirements, inducing the architecture with M_0 may appear to be more attractive as when compared to inducing the architecture with M_1 (using either JBoss or WLS). This is because M_1 incurs license costs for WLS. Moreover, looking at S_1 when induced with JBOSS, S_1 is likely to be in magnitude

slower than S_0 as when induced with JacORB. This means that S_1 (JBOSS) will support fewer TOPS and consequently will create less value added per second as when compared to S_0. For this low load, the fault-tolerance and load-balancing services need not be implemented on S_0. If options analysis is not used, M_0 will be a no-brain choice for inducing the Duke's Bank architecture. Though inducing the architecture S_1 with M_1 (using WLS) appears less attractive than M_0 (JacORB), S_1 may still carry *embedded growth options* which will only materialize if the load grows. If we use a PV or DCF approach, the resulted valuation will compute the present value as realized and ignore these growth options. In other words, inducing the architecture with WLS if undertaken, PV or DCF would hint that S_1 would destroy value rather than create it. Formulating this argument, a PV approach, for example, will leave us with $ValueS_1$ = PV. However, $ValueS_1$ is actually $ValueS_1$ = PV + *Opt*. That is, M_1 carry embedded growth options, Opt. The *Opt*, if left unexercised, are ignored by the non-options analysis. Hence, *Value* for S_1 is then said to be underestimated. As a result, S_0 may look more attractive (Table 6.16a). The PV and DCF calculation of Table 6.16a shows that S_1 is the least attractive for this range of load. The computation is based on the benefits of supporting 100 TOPS less their costs. However, the computation ignores the growth options on S_1 in supporting additional 632 TOPS using the first host. Similarly, the PV and DCF systematically undervalue the growth potential of S_1 (Jboss) and S_0 (JacORB) in respectively supporting 300.26 TOPS and 446.26 TOPS. In other words, PV and DCF ignore the flexibility value of S_1 and S_0 in responding to the growing load at host 1. Note that it is a fact that NPV or DCF does not work well for projects with future decisions that depend on how uncertainty resolves. Though they can be used to evaluate the operational benefits in a stable environment with well-understood and measurable costs and benefits, they have little to offer when capturing additional value due to flexibility under uncertainty, such as strategic opportunities and the ability to respond to changing conditions. Using PV or DCF, S_1, when induced with WLS, reports negative values upon inducing the architecture with WLS for this range of load. However, the situation indicates that these results underestimate the value of S_1, as S_1 can better respond to uncertainty, where the load is likely to grow over 100 TOPS. In Table 6.16b, we have turned to the intuition and used ArchOptions to capture the growth options on S_1 and S_0. The volatility parameter

is an expression of the range of "benefits" at a host. For example, consider S_1 (WLS): the benefits could "wander" from zero (i.e., idle state with no operations executing at a second) to the benefits derived from full utilization of capacity (i.e., in the support of 732 TOPS). That is, the volatility of 66% for S_1 (WLS) indicates that the benefits of executing the TOPS is in the range of \$0(idle) to \$92.42(full utilization) per second on host 1. Similarly, for S_0 (JacORB): the 45% volatility for S_0 (JacORB) indicates that the benefits of executing the TOPS are in the range of \$0(idle) to \$69.04 (full utilization) per second on host 1. As far as the options on S_1(WLS) are concerned, S_1 has "pulled" the options on one host for this range of load. This is because we have accounted for the possible fluctuation in the derived values from supporting the TOPS. Considering such "fluctuation" provides us with better insights on the architectural potential of S_1 in support of this likely change in load. Table 6.16b suggests S_1 has reported a value added of \$0.017 on 1 host.

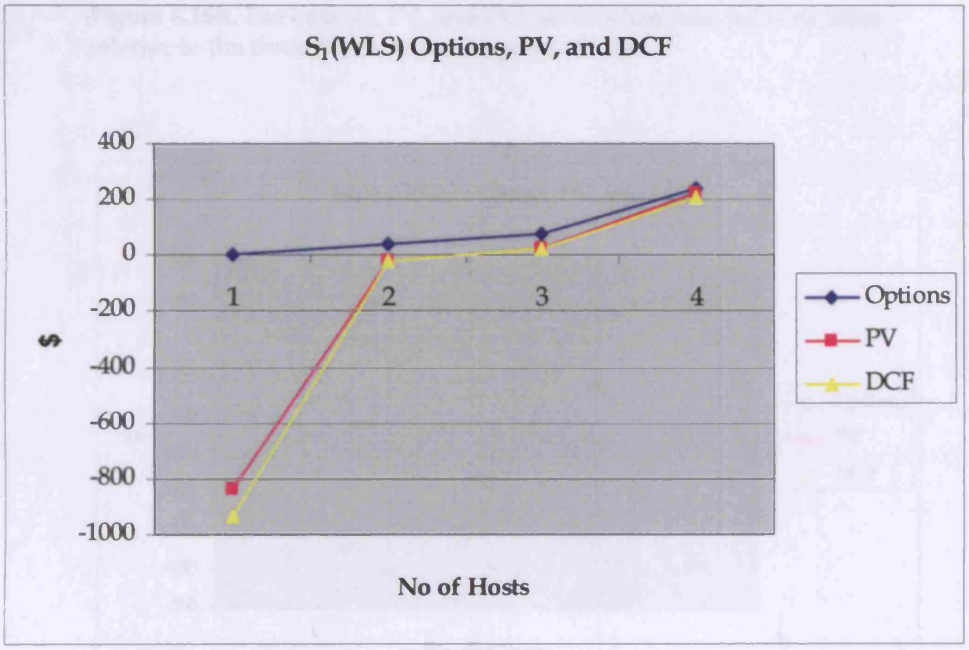


Figure 6.16a. The options, PV, and DCF on S_1 when induced with WLS relative to the throughput valuation point of view

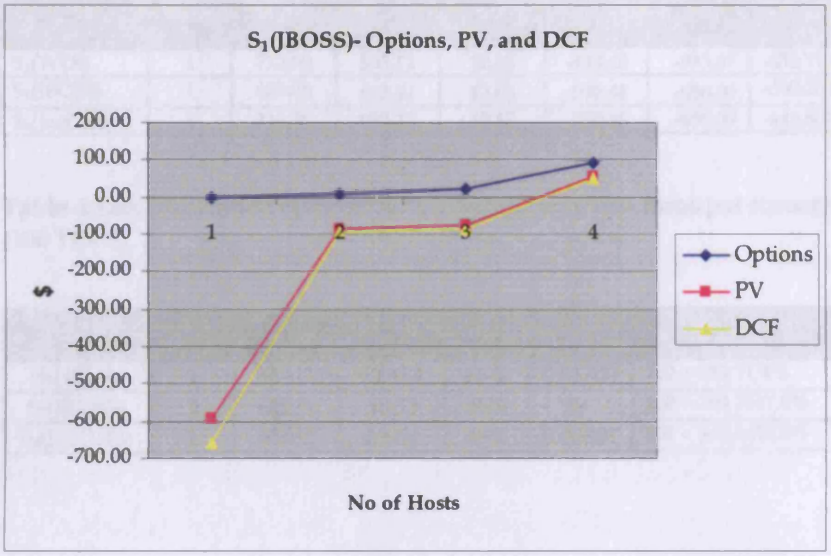


Figure 6.16b. The options, PV, and DCF on S_1 when induced with JBoss relative to the throughput valuation point of view

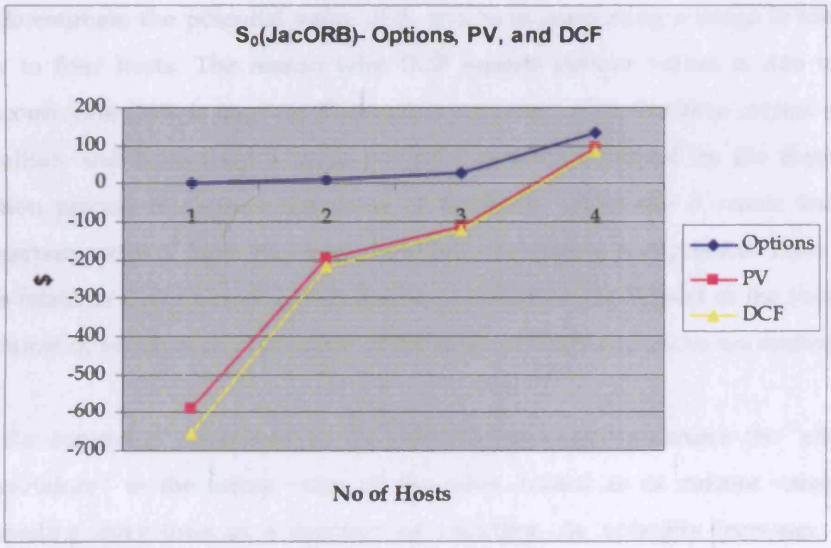


Figure 6.16c. The options, PV, and DCF on S_0 when induced with JacORB relative to the throughput valuation point of view

Table 6.16a. Illustration NPV and DCF per second ($) very low throuput scenario (100 TOPS)

100 TOPS	No Hosts	Max TOPS	C	X	PV	DCF	Value Ignored (TOPS)
S_1(WLS)	1	732.00	853.11	12.63	-840.48	-933.87	-632 TOPS
S_1(JBOSS)	1	400.26	603.11	12.63	-590.48	-656.09	-300.26TOPS
S_0(JacORB)	1	546.80	603.11	12.63	-590.48	-656.09	-446.80TOPS

Table 6.16b. Illustration options per second ($) very low throuput scenario (100 TOPS)

100 TOPS	No Hosts	C	X		Options	Actual Value (TOPS)
S_1(WLS)	1	853.11	92.424	66%	0.01700	100 + 632 TOPS
S_1(JBOSS)	1	603.11	50.53	35%	0+	100 + 300.26TOPS
S_0(JacORB)	1	603.11	69.04	49%	0.00001	100 + 446.80TOPS

Observation 4: Comparing PV and Options: the impact of volatility on value

A critical difference between PV/DCF and real options is the effect of uncertainty (or risk) on value. Figures 6.16a-c shows that PV and DCF systematically underestimate the potential value of S_1 and S_0 in supporting a range in load on one to four hosts. The reason why DCF reports steeper values is due to the discount rate (10% is used for illustration purposes only). We have turned to the intuition and have used a more powerful technique offered by the theory of option pricing to capture the value of flexibility under the dynamic and the uncertain range of load. However, how this uncertainty is expressed? How does this relate to Duke's case? Let us have a close look at the impact of the volatility parameter, which is an expression of the value of flexibility under uncertainty.

In the context of ArchOptions, the volatility parameter estimates the "cone of uncertainty" in the future value of the asset, rooted as its current value and extending over time as a function of volatility. As volatility increases, total uncertainty around the benefits also increases. The more TOPS a host is likely to support, the more likely that the actual benefits to "wander" up and down and deviate from the expected present value if the load grows. Hence, the more volatile the environment is said to be.

Let us now assume that Duke's Bank needs to support more customers. Assume that the load is likely to grow and be in the range of 600- 686 TOPS (Table 6.17a): S_1, when induced with WLS, realizes the change in load by one host. S_0, when induced with JacORB, will need two hosts and will incur the cost of developing the fault-tolerance and load-balancing services on the structure. Yet, S_1 when induced with JBoss will require three hosts and will incur additional hardware costs for completing the 686 TOPS. Figure 6.17 shows a scenario for a likely load of 600-686 TOPS for S_1 when induced with WLS and for S_0 when induced with JacORB. S_1 could be regarded as an investment with a wide range of possible outcomes. However, S_0 is an investment with a relatively narrower range. For S_1, the investment is said to be more volatile. This is because S_1 can support more TOPS/host resulting in a possible range of values. Relating this to PV, this means that there is a chance of producing positive PV in the future. Hence, a real option under this set of outcomes would have value. As for the S_0, the valuation under this scenario is more stable. This is because S_0 can support at most 686 TOPS for the existing configuration. This means that S_0 has no chance of producing a project with a positive NPV beyond 686 TOPS. That is an option using the latter set of outcomes would have no value.

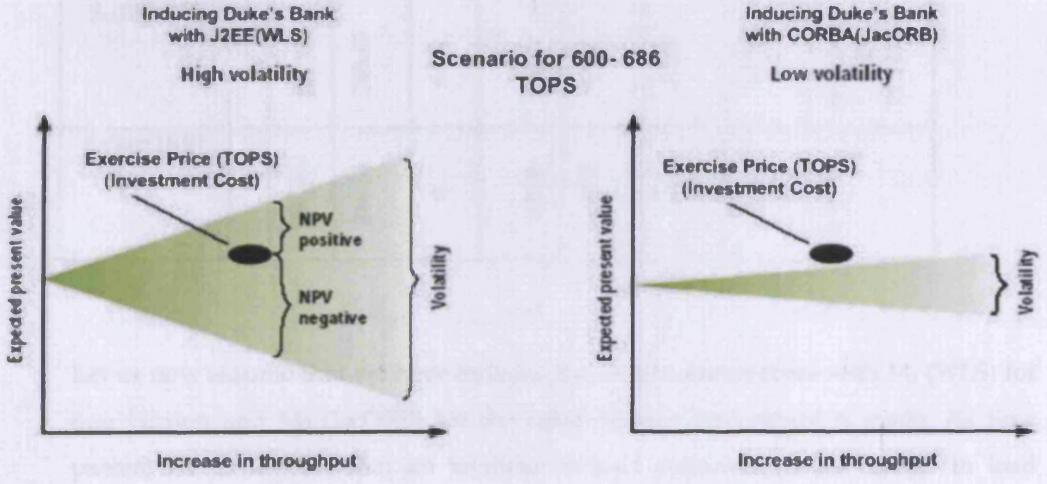


Figure 6.17. Impact of volatility on value

Table 6.17a. PV and DCF ($) per second for supporting 686 TOPS on S_0 and S_1 and the values they ingnore

[illegible]	No Hosts	Max TOPS	[illegible]	[illegible]	PV	DCF	Value Ignored (TOPS)
S_1(WLS)	1	732	124.36	216.54	92.18	83.80	-46 TOPS
S_1(JBOSS)	3	763	193.51	216.54	23.03	20.93	-77 TOPS
S_0(JacORB)	2	686	285.32	216.54	-68.78	-76.42	0 TOPS

Table 6.17b. Adjusted PV and the options in ($) per second under full utilization scenario of hosts for load greater than 686 TOPS on S_0 and S_1 and the values added per second

Full Utilization	No Hosts	[illegible]	[illegible]	Option	PV prior Adjustment	Adjusted PV	Options Predicted(S)	Added Value I (S)	Actual Value (TOPS)
S_1(WLS)	1	124.36	231.06	10.52%	92.18	106.7	106.7	14.52	686 TOPS Plus 46 TOPS
S_1(JBOSS)	3	193.51	240.85	6.9%	23.03	47.34	47.34	24.34	686 TOPS Plus 77 TOPS
S_0(JacORB)	2	285.32	216.54	0	-68.78	-68.78	0	0	0 TOPS

Let us now assume that we have induced the Duke's architecture with M_1 (WLS) for one version and M_0 (JacORB) for the other. Hence, investment is made. As time passes, let us assume that an increase in load materializes. As change in load materializes, uncertainty is assumed to be resolved. Thus, the present value, as a result of supporting more TOPS (analogous to the future value of a stock), can be then calculated more accurately. If we examine the PV of this scenario, we can see that PV reports $92.18/second for WLS for 686 TOPS. That is, this is equal to the

benefits minus the costs of completing the 686 TOPS. However, this value is said to be underestimated, as it ignores the additional 46 TOPS that S_1 can support using one host (i.e., 732 minus 46 TOPS). S_1, when induced with JBoss, reports a PV of $23.03, ignoring the additional value of supporting 77 TOPS for this configuration. S_0, when induced with JacORB, reports a negative PV. The negative value is attributed to cost incurred upon the development of the fault tolerance and the load balancing services on S_0. Let us now turn to options: Table 6.17b suggests that for 686 TOPS, S_1, when induced with WLS, creates more options than S_0 using one host. In particular, S_1 (WLS) reports a value of $106.7. S_1 (JBoss) reports a value of $47.3. S_0 (JacORB) reports a value of $0. Why is this difference? Technically speaking, this is because of the volatility parameter that captures variation in the value potentials of the said structures. For S_1 (WLS), the difference for S_1 (WLS) is attributed the range of possible returns that the additional 46 could ascribe to S_1(WLS). This means that for S_1 (WLS), the additional future values, if the range in load changes, is in the bound of $0(i.e., at most 686 TOPS) to $46*216.54/686(i.e., assuming equal returns upon supporting the additional 46TOPS). This will leave us with a volatility of %10.52, using the standard deviation of the returns over this bound. Similar argument applies for S_1 (JBoss), leaving us with a volatility equal to %6.9 in support of the additional 77 TOPS. S_0 (JacORB) reports $0 options. This is because S_0 (JacORB) cannot support additional TOPS on this structure. In the language of options, S_0 (JacORB) is not volatile and ceases to create options beyond 686 TOPS; henceforth, the reported zero values.

Let us now turn to PV again and assume an additional load has materialized (i.e., uncertainty has been resolved). Let us adjust the PV based on the new information at hand: if we compute the PV of the additional 46 TOPS for S_1 (WLS), this will leave us with an added value of $14.52 over the previously computed PV, as reported in Table 6.17b. If we compute the PV of the additional 77 TOPS for S_1 (JBoss), this will leave us with an added value of $24.34 over the previously computed PV for S_1(JBoss)- see Table 6.17b. Adjusting the PV, we sum these values with the previously reported PVs of Table 6.17b. This will leave us with $106.7 value for S_1 (WLS) and $47.3 value for S_1 (JBoss). Henceforth, this is a match with the ArchOptions results for S_1 (WLS) and S_1 (JBoss).

This observation leaves us with following conclusions: First, though it is still possible to adjust PV or DCF techniques for capturing the options, ArchOptions provides us with a ready and closed-form solution, rooted in options theory, for capturing the value of flexibility under uncertainty on a given architecture. This solution is said to be superior to PV and DCF, as the latter they systematically underestimate the value of the flexibility of an architecture under uncertainty. Secondly, the analysis of matching the adjusted PV values with that of ArchOptions confirms the correctness and the effectiveness of the model. Nevertheless, the effectiveness of ArchOptions is essentially rooted in our use of Black and Scholes options theory. The analysis, however, has established confidence on both its correctness and effectiveness. Third, the results of this observation show that the volatility parameter is critical for the valuation of the options. In real situations, the performance analyst/architect may inspect available performance benchmarks, screen historical load-trends to predict future ones, or use prototypes of partial implementations to collect performance indices. Consequently, volatility can be then empirically extracted. The analyst can make use of the sensitivity analysis we have provided in Chapter 4 for better understanding of the impact of throughput on the value added when uncertainty in the likely future load dominates.

Observation 5. Selecting a stable architecture

> The change impact analysis has shown that the architectural structure of S_1 is left intact when the scalability change needs to be accommodated. However, the structure of S_0 has undergone some changes, mostly on the architectural infrastructure level to accommodate the scalability requirements. From a value-based perspective, the search for a potentially stable architecture requires finding an architecture that maximizes the yield in the added value, relative to some future changes in requirements. As we are assuming that the added value is attributed to flexibility, the problem becomes selecting an architecture that maximize the yield in the embedded or adapted flexibility in a software architecture relative to these changes. Even, if we accept the fact that modifying the architecture or the infrastructure is the only solution towards accommodating the change, valuation the impact of the change becomes necessary to see how far we are expending to "re-maintain" or "re-

achieve" architectural stability relative to the change. Note that the economic interplay between evolving requirements, the flexibility of the architecture to accommodate the change, the structural impact, and the corresponding cost/value implications is the key towards selecting a "more" stable architectures that tends to add value as the requirements evolve. Though it might be appealing to the intuition that the "intactness" of the structure is the definitive criteria for selecting a "more" stable architectures, the practice reveals a different trend; it nails down to the potential added value upon exercising the change.

If you consider the case of S_0 and S_1 in response to the change in scalability for one host (Table 6.13a), the flexibility has yielded a better payoff for S_1 than for S_0, while leaving S_1 intact. This implies that inducing the Duke's Bank software architecture with M_1 is likely to be more stable relative to the future change in scalability, than when induced with M_0. However, the situation and the analysis have differed upon varying the number of hosts and upon factoring a license costs for S_1. Though S_0 has undergone some structural changes to accommodate the change, the case has shown that it is still acceptable to modify the architecture and to realize added value under the conditions that $U_nS_0 >>= U_nS_1$ for 7 or more hosts (Table 6.13c, Figure 6.12). Hence, what matters is the added value upon either embarking on a "more" flexible architecture, or investing to enhance flexibility which is the case for implementing load balancing and fault tolerance on S_0. For the case of WebLogic, Though M_1 is in principle more flexible, the flexibility comes with a price, where the flexibility turned to be a liability rather than a value for 7 or more hosts, as when compared with the JacORB, under the condition that $U_nS_0 >>= U_nS_1$. The case verifies our claims that the value of flexibility can guide towards the selection of architectures that tend to add more value, as the requirements evolve. These architectures have the potential of being potentially stable.

The analysis of the throughput valuation point of view, taking the throughput as a critical measure, has revealed a different trend upon taking into account the distribution cost and the added value of the supported TOPS

on a host. Deciding on a particular middleware to induce the software system architecture can be seen as an investment to purchase future growth options that enhance the upside potentials of the structure. Looking at the throughput valuation point of view, part of the growth options come from the ability of the induced-architecture to support more TOPS while minimizing the cost of distribution; henceforth, creating more options. These growth options are correlated with the TOPS that could be supported on a host and their exercise price. However, the choice is not straightforward as the future load is "dynamic" and uncertain. The range in which the load may change determines the suitability of the choice. If the likely load tends to be high and uncertain, an induced-architecture, which is volatile and holds more options, will be a favorable choice. If the range in the load is deterministic but low, the maintainability point of view may steer the selection (see Observation 3). In this regard, one could characterize the choice of a "more" stable architecture as a multi-objective optimization activity in which one trades maintainability for performance. In real situations, selecting a stable architecture implies finding an architecture, which maximizes the yield in the added value relative to the two valuation points.

The options analysis has complemented the structural and the behavioral analyses to quantify the impact of the change on the software architecture. The intuition is that complementing both structural and behavioral impact analysis with a value-based calculation, the combination provides the architect/analyst with a useful tool for understanding extent to which the software system tend to be flexible relative to a likely change in requirements, a cost/value indictors of the impact of the change on the structure, its performance which is directly linked to value, the likely success (failure) of the software system evolution, and consequently the potential stability of the software architecture relative to the change.

6.3.8 Implications on the Discipline

In subsequent sections, we draw some preliminary lessons and insights that have derived upon the application of ArchOptions. This could stimulate future research in the area of relating requirements to software architectures and consequently advance our understanding to the architectural stability problem, when addressed from the evolution of the non-functional requirement perspective.

Implication 1. Understanding architectural stability has to be in connection with the solution domain

Our hypothesis that middleware induced-software architectures differ in coping with changes is verified to be true for the given change. Based on the pervious observations, we can see that the stability of S_1 and S_0 appears to be dependent on the flexibility of the middleware in accommodating the likely changes in the scalability requirements. For the category of distributed software systems that are built using middleware, the results of the case study affirm the belief that investigating the stability of the distributed software architecture could be fruitless, if done in isolation of the middleware, where the middleware constraints and dominates much of the solution that relate to the non-functionalities, managing system resources, and their ability to smoothly evolve over the life time of the software system. Hence, the development and the analysis for architectural stability and evolution shall consider the "coupling" between the architecture and the middleware. This addresses pragmatic needs and is feasible even at earlier stages of the software development life cycle: a considerable part of the distributed system implementation could be available, when the architecture is defined, for example, during the Elaboration phase of the Unified Process. We also note that the change in requirements could have been addressed by other architectural mechanisms. However, the middleware has guided the solution for evolving the software system. For instance, the choice of replication as an architectural mechanism for scaling the software system, with a given architectures S_1 and S_0 was respectively guided by the clustering primitives provided by M_1 and the core capabilities provided by M_0 to support load balancing and fault tolerance. Interestingly, Di Nitto and Rosenblum [1999] state that "despite the fact that architectures and middleware

address different phases of software development, the usage of middleware and predefined components can influence the architecture of the system being developed. Conversely, specific architectural choices constrain the selection of the underlying middleware used in the implementation phase". In more abstract terms, Rapanotti, Hall, Jackson, and Nuseibeh [2004] advocate the use of information in the solution domain (e.g., the middleware-to be induced for our case) to inform the problem space:

> "Whereas Problem Frames are used only in the problem space, we observe that each of these competing views uses knowledge of the solution space: the first through the software engineer's domain knowledge; the second through choice of domain-specific architectures, architectural styles, development patterns, *etc*; the third through the reuse of past development experience. All solution space knowledge can and should be used to inform the problem analysis for new software developments within that domain" [Rapanotti et al., 2004].

The "coupling" between the middleware and the architecture becomes of higher interest in case of developing and analyzing software systems for evolution. This is because the solution domain can guide the development and evolution of the software system; provide more pragmatic and deterministic knowledge on the potential success (failure) of evolution, and consequently assist in understanding the stability of the software architectures from a pragmatic perspective.

Implication 2. Understanding architectural stability: intertwined with changes in non-functional requirements, style, and the middleware

> Following the definition of Shaw and Garlan [1996], a *style* defines a set of general rules that describe or constrain the structure of architectures and the way their components interact. Styles are a mechanism for categorizing architectures and for defining their common characteristics. Though S_1 and S_0 have exhibited similar styles (i.e., three-tier), they have differed in the way they cope with the change in scalability. The difference was not only due to

the architectural style, but also due to the primitives that are built-in in the middleware to facilitate scaling the software system. The governing factor, hence, appears to be to a large extent dependent on the flexibility of the middleware (e.g., through its built-in primitives) in supporting the change. The intuition and the preliminary observations, therefore, suggest that the style by itself is not revealing for the stability of the software architecture when the non-functional requirements evolve. It is, however, a factor of the extent to which the middleware primitives can support the change in non-functional requirements. Interestingly, Sullivan et al. [1997] claims that for a system to be implemented in a straightforward manner on top of a middleware, the corresponding architecture has to be compliant with the architectural constraints imposed by the middleware. Sullivan et al. [1997] support this claim by demonstrating that a style, that in principle seems to be easily implementable using the COM middleware, is actually incompatible with it. Following a similar argument, adopting an architectural style that is in principle appear to be suitable for realizing the non-functionality and supporting its evolution, may not be complaint with the middleware in the first place. And if the architectural style happens to be compliant with the middleware, there are still uncertainties in the ability of the middleware primitives to support the change. In fact, the middleware primitives realize much of the non-functional requirements. Hence, the architectural style by itself may not be revealing for potential threats that the architecture may face when the non-functional requirements evolve. The evolution of non-functionality maybe in principle easily supported by the style, but could be uneasily accommodated by the middleware. An observable advantage of scaling the software architecture induced by S_1, for example, is that no development effort required to realize the scalability requirements through replication, as when compared to that of S_0, knowing that in principle the style of S_1 and S_0 exhibit similar capabilities.

Engineering for stability and evolution, requirements engineering has not only to be aware of the architecture (e.g., the style), but also of the underlying middleware. For example, if we take a goal-oriented approach to requirements engineering (e.g., [Dardenne et al., 1993]), we advocate

adjusting the non-functional requirements elicitation and their corresponding refinements to be aware of both the architectural style and the constraints imposed by middleware. The operationalization of these requirements in the software architecture have to be guided by both the architectural style, the complaint middleware for the said style, and guided by previous experience. This, we believe, is a pragmatic need towards engineering requirements and developing "evolvable" software architectures that tend to be stable as the non-functional requirements evolve.

6.3.9 Concluding Remarks

We have used change in scalability, a representative critical change in non-functional requirements to steer the study and apply the model. We have appealed to the use of structural and behavioral analysis, combined with value-based analysis, to inform the tradeoff and select a "more" stable architecture. Though the reported observations reveal a trend that agrees with the intuition, research, and the state-of-practice, confirming the validity of the observations are still subject to careful further empirical studies. These studies may need to consider other non-functional requirements, their concurrent evolution, and their corresponding change impact on different architectural styles and middleware. As a limitation, we have relaxed considering the change impact of scaling up the software system on other non-functional requirements like security, availability and reliability. However, we note that the analysis might get complex upon accounting for the impact of the change on other non-functional requirements and their interactions. Note the change could positively or negatively impact other non-functional requirements and understanding the cost implications is not straightforward and worth a separate empirical investigation. In this context, utilizing the NFR framework [Mylopoulos et al., 1992] could be promising to model the interaction of various non-functional requirements, their corresponding architectural decisions, and the negative/positive contribution of the architectural decisions in satisfying these non-functionalities. The framework could be then complemented by means for measuring (i) the corresponding cost of implementing the change itself, and (ii) the additional cost due

to the impact of the change on other contributing or conflicting non-functionalities, as realized by either the CORBA or the J2EE middleware-induced architectures.

It is also worth noting that the investment decision in either CORBA or the J2EE might be influenced by other factors, such as the skills of the developers, the project maturity, and other organizational factors. The devised real options model does not explicitly take into account these factors. The treatment of these factors is left implicit and sufficiently addressed by our use of COCOMO II, where COCOMO II carries parameters to adjust the cost estimates based on these factors. It could be also argued that in iterative development, when estimations are continuously recalibrated (e.g., in the Unified Process), it is possible to come up with estimations that are more accurate than COCOMO II, as they will take into account the above mentioned factors.

We note that the flexibility of either solutions (i.e., the CORBA or the J2EE induced-architectures) is closely tied to the problem domain. In particular, domain-specific functional characteristics can also influence the flexibility of the solution and its behavior, as both the application component and the infrastructure are tightly coupled [Liu and Gorton, 2003]. The way the application components and the infrastructure are coupled varies across various middlewares. For this study, the functional characteristics are assumed to be stable for both the J2EE and the CORBA versions; that is, they have not undergone any changes that require from us understanding the impact of the functionality change on the flexibility of either solutions. It will be interesting, however, to investigate how changes in the domain functional characteristics can impact the flexibility and the stability of the middleware-induced architectures.

Under no considerations should the results be regarded as a definite distinction of the merit of one technology over the other, but yet still revealing on the scalability dimension. The reason is due to the fact that we have only used "flavors" of CORBA and J2EE, respectively through JacORB, JBoss, and WebLogic.

Table 6.18a and Table 6.18b relate the case to the method developed in the previous Chapter. The case has exemplified the valuation points of view framework that we have outlined in Chapter 5. It has appealed to the use of two: these are structural point of view (maintainability valuation point of view) and behavioral point of view (throughput valuation point of view). For Phase I of the method, the above case has adopted a goal-oriented approach to elicit the change as a goal that need to be achieved for scaling the structure. The refinement was done in relation to the middleware to be induced. For the throughput valuation point of view, we have assumed that we are given likely changes in load-range. We have attempted to relate the load-range to performance, which is an architectural quality, as a way to link the change to the architecture. Nevertheless, we could have adopted goal-oriented approaches for eliciting these ranges. However, the purpose of the case is to verify the thesis claims, illustrate the use of the model and simulate its steps; evaluate the maturity of model's interpretations and its applicability.

Table 6.18a. Relating the cases to Phase I of the method

Phase I	Case 2
Setting the objectives for evaluating architectural stability	Objective: Which middleware-induced architecture is more stable with respect to future changes in scalability and relative to two valuation points of view
Eliciting the change $\{i_1, i_2, \ldots, i_n\}$ that are critical to the set objectives	Maintaining scalability on the structure
	Likely ranges in load
Relating the change to the architecture	The change was refined and traced to the middleware primitives responsible in realizing scalability
	The change was related to performance

Table 6.18b. Relating the cases to Phase II of the method

Phase II	Case 2
Identify valuation points of view	Structural(Maintainability valuation point of view): Maintainability, configuration, and deployment
	Behavioral(Throughput valuation point of view): throughput
Identify the value of the architectural potentials with respect to the change	Structural(Maintainability valuation point of view): J2EE built-in primitives in realizing scalability through replication
	Behavioral(Throughput valuation point of view): value in supporting additional TOPS
Volatility	Structural: optimistic, likely, and pessimistic
	Behavioral(Throughput valuation point of view): return on possible values of supported TOPS in a range or modeling assumptions
Estimate the cost of accommodating the change	Structural(Maintainability valuation point of view): The cost of implementing scalability on each structure
	Behavioral(Throughput valuation point of view): Price/TOPS

6.4 Comparative Analysis

We evaluate ArchOptions using some general qualitative characteristics including simplicity of use, openness, comprehensiveness, and prediction effectiveness.

Qualitative Characteristics

The analogy that ArchOptions makes with options is simple, yet powerful and comprehensive enough to provide basis for analyses supporting plenty of problems. We have just utilized this simple and intuitive analogy to address two complex architectural centric-evolution problems: valuing the long-term cost-effectiveness of refactoring and informing the selection of more stable middleware. Further, in Chapter 7, we will highlight some possible unexplored uses of the model to reason about the worthiness of investing in restructuring "traditional" systems to support

aspect-orientation, with the objective of facilitating future maintainability and better stability.

ArchOptions is a composite model, for it is flexible to incorporate estimations based on both expert knowledge and parameterized models. For example, our use of COCOMO II to estimate C_{ei} and the use of subjective estimates of $x_i Vs$, based on the twin asset, uses both expert knowledge and parameterized models to estimation. Note that such a combination may result in higher estimate accuracy, as when compared to the use of models, which are solely based on expert knowledge or parameterized models. For example, for the case of the middleware selection, we have used TAO and benchmarks as twin assets. The intent behind using the twin asset is to understand the behavior of an option by using a corresponding replicating portfolio (i.e., a twin). The portfolio and the options are interchangeable for all practical purposes and should worth the same. The assumption is that the two assets, the option and the twin, with the same payoffs under same conditions, must have the same price. If we know how much the twin asset is worth in the present, we can then determine how much the option is worth in the present. The analogy of ArchOptions with options theory holds such assumptions, which we believe, is strength as it is grounded in a sound theory. Further, the use of the twin asset is said to theoretically complement software engineering approaches, which advocate using analogy to estimate cost in software (e.g., [Shepperd et al., 1996]) for improving the prediction.
A notable desirable feature of ArchOptions is its flexibility and openness; the model does not define rigorous ways for estimating its parameters, conducting its steps, and confirming specific actions to execute, following the options computation. Consequently, we note that evaluating methods like ArchOptions is rather hard, as their effectiveness is dependent on the way practitioners apply them. For example, practitioners may have to tailor ArchOptions to address the needs of a specific architectural-centric evolution problem and its desired stability requirements. In addition, the nature of the decisions made when applying ArchOptions fundamentally varies from one project to another, with the addressed problem, and across organizations. As a result, the effectiveness of its application is subject to the context in which the model is applied. ArchOptions is open; it could be easily integrated to complement existing architectural evaluation methods, highlighted in Chapter 2, with the objective of explicit evaluation for stability while taking an

economics-driven perspective. The integration may provide a basis for analyzing the complexity and economic ramifications of a change in requirements and its impact on the architecture and/or the associated architectural decisions.

Prediction Effectiveness

ArchOptions levels on a sound theory in financial engineering(Nobel Prize winning). The ArchOptions prediction is inherently effective as it is grounded in the use of Black and Scholes technique. Nevertheless, Observations 3 and 4 of Section 6.3.7 have confirmed the effectiveness of the prediction and the correctness of the computation through examples. The observations left us with the following conclusions: First, though it is still possible to adjust PV or DCF techniques for capturing the options, ArchOptions provides us with a ready and closed-form solution, rooted in options theory, for capturing the value of flexibility under uncertainty on a given architecture. This solution is said to be superior to PV and DCF, as PV and DCF systematically underestimate the value of the architectural flexibility under uncertainty. Secondly, the analysis and our ability to match the adjusted PV values with that of ArchOptions (refer to Observation 4) confirms the effectiveness of the model.

To further confirm this claim and extend the confidence in the model prediction, we have conducted three small comparative exercises. In the first exercise, we report on the student's experience in implementing the structural scalability change on the Duke's architectures. We report on how the actual value is compared to that of the ArchOption's predicted one. In the second exercise, we have benchmarked some representative results of the refactoring case against the binomial options model of [Cox and Rubinstein, 1985], one of the most cited options techniques in the economic literature. In the third exercise, we have compared the ArchOptions results of the refactoring case to that of [Leitch and Stroulia, 2003], where the latter is based on cost/benefit analysis.

In conducting the above exercises, we have used the Magnitude Relative Error (MRE) [Conte et al., 1986], a commonly used measure, for the evaluation of estimation models. The objective is to evaluate the effectiveness of the ArchOptions

prediction and to understand the degree of deviation of the estimated options to that of the actual ones. The tailored MRE for our case is given in the below equation (6.5):

$$MRE = \left| \frac{Options_{actual} - Options_{predicted}}{Options_{actual}} \right| \quad (6.5)$$

$$\text{Such that } Options_{actual} > 0$$

One of the motivations behind using real options theory is because the value of the architectural potential to the change is uncertain as the change is uncertain. Let us assume that uncertainty is resolved: the value becomes certain. We can then calculate the value added, $Options_{actual}$ of (6.5), using PV. ArchOptions is then used to calculate the $Options_{predicted}$. Using the $Options_{actual}$ and $Options_{predicted}$, we could then calculate the MRE. We use the prediction level Pred(*l*) of equation (6.6). This measure is often used in the literature and is a proportion of the observations for a given level of accuracy.

$$Pred(l) = K/N \quad (6.6)$$

Where, N is the total number of observations, and K is the number of observations with an MRE less than or equal to *l*. A common value for *l* is 0.25. The Pred (0.25) gives the percentage of observation that were predicted with an MRE equal or less than 0.25. Conte et al. [1986] suggest an acceptable threshold value for the mean MRE to be less than 0.25. For Pred(0.25), Conte et al. [1986] suggest an acceptable threshold value to be greater or equal to 0.75.

Exercise 1. Using the help of a student, the Duke's bank was implemented. All effort was made to ensure that the student mimics the twin asset and utilize the guidelines provided by the supporting documentation for implementing and "switching on" scalability on each structure. SLOC were gathered from the corresponding implementation. The student implementation of the load balancing and the fault tolerance services on S_0 (JacORB) yielded to 12226 SLOC in contrary to the estimated 9240 SLOC. The 12226 SLOC corresponds to costs ranging from $127659(optimistic)

to $199470(pessimistic) according to Table 6.19a. These figures are computed using COCOMOII and based on similar computation assumptions to that of Table 6.13a and Table 6.13b. This means that if the student would have used S_1 (JBOSS), then savings in person-months relative to S_0 (JacORB) would have been realized. These savings, $x_i V_{PM}$ S_1 (JBOSS), are in the range of $126101.8 to $197035.3 and according to Table 6.19b.

Using PV, we have computed Options $_{actual}$, on the maintainability valuation points of view, and based on the assumptions that value and cost are certain (i.e., as the architectural potential is now certain). For Options $_{predicted}$ of (6.5), we have used the options results of Table 6.13a and 6.13b. Using equation (6.5), the reported variation is in an acceptable range with 24% MRE.

Table 6.19a. The SLOC and the corresponding cost of implementing the load balancing and fault tolerance by the student on S_0(JacORB)for one host (Maintainability valuation point of view)

	SLOC		C... S (JacORB)
Maintainability valuation point of view	12226	Optimistic	127659.8
		Likely	159575.8
		Pessimistic	199470.4

Table 6.19b. The predicted options ($), PV ($), and MRE on S_1 (Jboss) relative to S_0 (JacORb) relative to the maintainability valuation point of view

		C... S (Jboss)	V... S (Jboss)	ArchOptions	Student (PV)	MRE
Maintainability valuation point of view	Pessimistic	1158	126101.8	94892	124543.7	0.238
	Likely	1948	157627.7	118615	155679.7	0.238
	Optimistic	2435	197035.3	148269	194600.3	0.238

The deviation, however, could be attributed to the following reasons: the "unfaithfulness" that the student may have shown to the twin asset, TAO; his programming skills and style; the code optimization; any probable implementation defects; and so forth. As a limitation, we acknowledge that the sample is too small to generalize a conclusion. Replicating this trial, during the PhD period, was difficult for two major reasons: First, the experiment is time and human demanding; it is

difficult to accommodate within the doctoral period. Second, the experiment includes some variables, which were difficult to control. The skills of the developer, the correctness and the completeness of the code, the programming style, are just a few variables to enumerate. The conducted study, however, provides the promise for future replication. As future work, we aim to conduct a careful systematic study, possibly by assigning this exercise to a large group of students of advanced or graduate standing(with strong programming and distributing software engineering skills) to empirically arrive at a better insight on the predictive effectiveness of ArchOptions in relation to these variables.

Exercise 2. One of the most cited options techniques in the economic literature is the binomial options model of [Cox and Rubinstein, 1985]. In brief, this binomial model assumes that the value of the underlying asset (in the ArchOptions case, denoted by x_iV) follows a binomial distribution. Starting at time zero, in one time period t, x_iV may rise to u x_iV with probability q or fall to d x_iV with probability $1-q$, where $d<1$, $u>1$, and $d<r<u$. In contrast, the use of Black and Scholes [1973] assume that x_iV is lognormaly distributed. Both assumptions means that the value of the underlying asset can increase to infinity, but only fall to zero [Hull, 1997]. The terminal value of a call option under [Cox and Rubinstein, 1985] at T is given by equations (6.7):

$$Cu = \max [0, u\, x_iV - C_{ei}] \quad \text{and}$$

$$Cd = \max [0, x_iV - C_{ei}],$$

with probabilities q and $1-q$, respectively (6.7)

We have cast the ArchOptions model to use the options valuation technique of [Cox and Rubinstein, 1985]. For 18 observations, we have assumed that we are given values for u and d. Given u and d, we have calculated the "rise" and the "fall" in the values of the architectural potential in response to change for a time period. Let us now assume that the computed options using [Cox and Rubinstein, 1985] correspond to $\text{Options}_{\text{actual}}$. Using a tool accompanied with [Hull, 1997], we have approximated the volatility from the possible ranges of the probability-adjusted values, arriving at the so-called implied volatility. The implied volatility and the corresponding

behavior of the adjusted-probability value are therefore comparable. Using the implied volatility, we can then use ArchOptions to estimate the $\text{Options}_{\text{predicted}}$. Table 6.19a reports on the MREs of the results for 18 observations. Figure 6.18 shows a very tiny variation upon the computation of the ArchOptions calls using Binomial theory [Cox and Rubinstein, 1985] and [Black and Scholes, 1973].

For Pred(0.25), ArchOptions reports 95% accuracy, which is in an acceptable accuracy range in accordance to [Conte et al., 1986]. The results, as sketched in Figure 6.18 and depicted in Table 6.20, extend the confidence in the ArchOptions prediction. The variation, however, could be attributed to the following reasons: First, the assumptions that the Binomial options theory makes to the computation. Second, the approximation of the implied volatility from the probability adjusted values. However, the application of Black and Scholes [1973] offers a closed and an easy-to-compute solution, for it assumes that $x_i V$ is lognormaly distributed, not requiring $x_i V$ to be probability-adjusted for rise and drop in value, as when compared to [Cox and Rubinstein, 1985]. Furthermore, determining u and d is a difficult empirical problem because asset "price" rarely follow a classical binomial process [Hull, 1997].

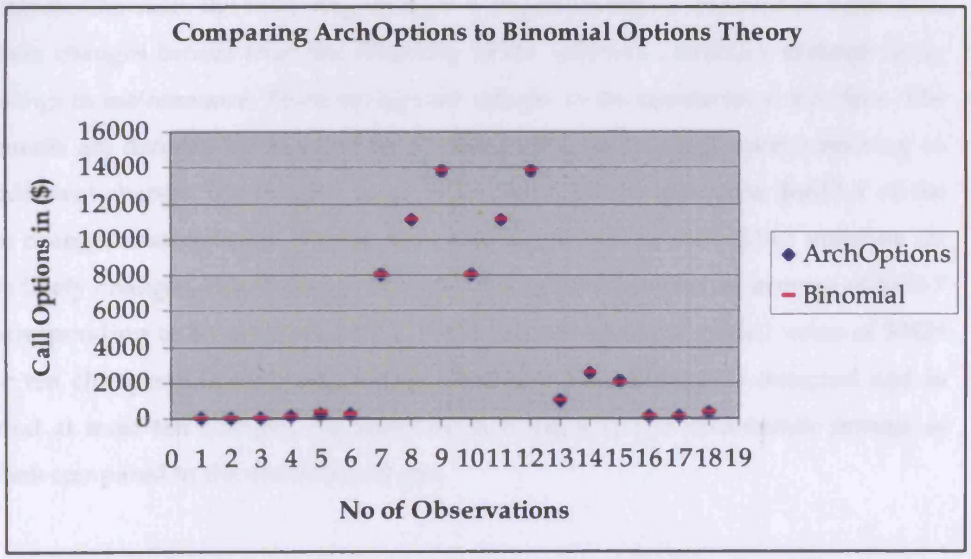


Figure 6.18. ArchOptions and Binomial options compared for 18 observations

Table 6.20. The Refactoring case study: the MRE upon computing the calls of ArchOptions using [Black and Scholes, 1973] and [Cox and Rubinstein, 1985]

Observation	C	X	T	ArchOptions	Binomial	MRE
1	38296	3516	4	0	0	0
2	47834	4396	4	0	0	0
3	59795	5494	4	0	0	0
4	29953	8046	4	100.6	92.5	0.0875676
5	36458	11039	4	204	281	0.2740214
6	45609	13764.1	4	252	236	0.0677966
7	1893	9938.9	0.5	8046	8045	0.0001243
8	2366	13405	0.5	11039	11039	0
9	2958	16722	0.5	13764	13765	7.265E-05
10	1893	9939	4	8052	8046	0.0007457
11	2366	13405	4	11045	11039	0.0005435
12	2958	16722	4	13772	13764	0.0005812
13	1893	2877.6	4	993	992	0.0010081
14	2366	4887.7	4	2522	2521	0.0003967
15	2958	4983	4	2029	2026	0.0014808
16	1893	1858	3	71.9	71.8	0.0013928
17	2366	2323	3	90.2	90.03	0.0018883
18	2958	3201.5	3	298	320	0.06875

Exercise 3. We compare some of the ArchOptions results for the refactoring case of Section 6.3 to that of [Leitch and Stroulia, 2003], where the latter is based on PV analysis. Consider the following changes in requirements as depicted in Table 6.21. These changes benefit from the flexibility of the refactored structure through likely savings in maintenance. These savings are relative to the unrefactored structure. The benefits are denoted by X_{iv}s and based on accumulated savings upon exercising an additional change. The benefits range from $464.6 for one change to $4640 if all the ten changes materialize in a given time, leaving us with us with %14.1 volatility for ten likely changes. Every change is made, it is assumed to cost an average of $181.7 corresponding to an estimate for C_{ei}. ArchOptions reports an added value of $2823 for ten changes following refactoring. That is, if refactoring was designed and in mind at most ten changes, the structure is worth $2823 of man-month savings as when compared to the unrefactored one.

Let us now assume that uncertainty is resolved: this means the value of the structure is certain. PV can be then used. Using options analysis with the assumption that uncertainty is resolved: for the one to ten changes, the options held on the

architecture are worth $2823. That is, if we calculate the value of the structure now for any change using options thinking, we are left by PV for the change + Growth Options = $2823 as shown in Table 6.21.

Table 6.21. Comparing ArchOptions to [Leitch and Stroulia, 2003]

Changes	X	Options Now	Stroulia
1	464.6	2823	-1352.4
2	929.2	2823	-887.8
3	1393.8	2823	-423.2
4	1858.4	2781.5 + 41.4	41.4
5	2323	2315.5 + 506	506
6	2787.6	1817 + 970.6	970.6
7	3252.2	1387.8 + 1435.2	1435.2
8	3716.8	923.2 + 1899.8	1899.8
9	4181.4	459 + 2364	2364.4
10	4640	2823	2823

Let us now turn to [Stroulia and Leitch, 2003]. The results show that their use of PV underestimates the value of the structure as they ignore the growth options held on the architecture. For example, for 1 to 4 changes, they report negative values for less than 4 changes. That is, if a decision need to be made based on PV, the investment in refactoring may seem to be unattractive ignoring the growth options held on the architecture. Only for ten changes, Stroulia and Leitch's use of PV reveals the $2823 options value, which the actual value of the structure and as shown in Figure 6.19.

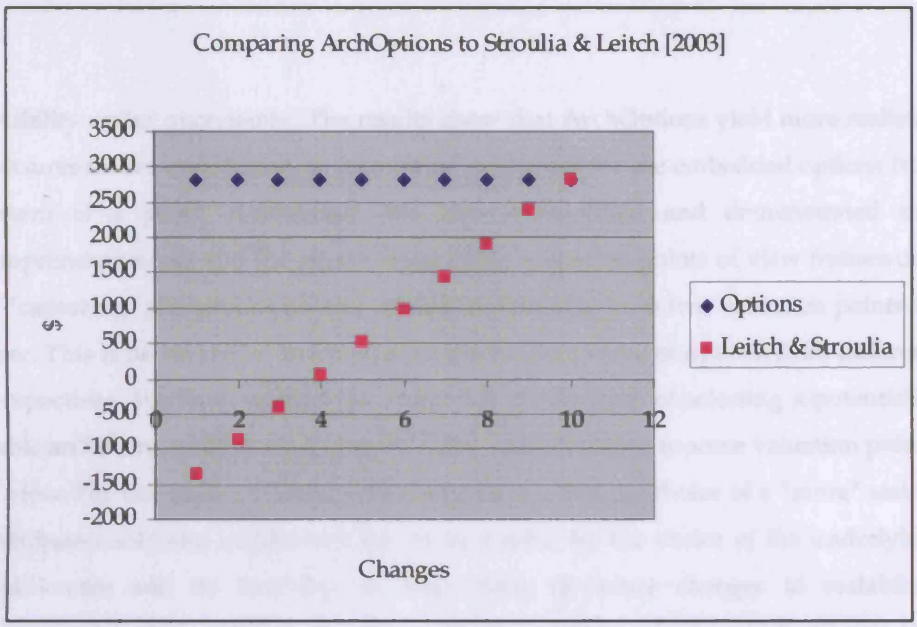


Figure 6.19. Comparing ArchOptions to [Leitch and Stroulia, 2003]

6.5 Summary and Implications

The evaluation has explored the approach "fitness" in addressing two architectural-centric software evolution problems. In the first case, we have taken refactoring, as a representative example, to show how ArchOptions can be used to assess the worthiness of re-engineering an architecture for change. The importance of this example is not in the architecture itself, but in how we have used the theory and the model to reason about the flexibility of the architecture in relation to likely change in requirements. We have verified the claim that the flexibility of an architecture in face of likely changes has values in the form of real options. In the second case, we have shown how ArchOptions can inform the selection of a "more" stable middleware-induced software architecture in the face of future changes in non-functional requirements, taking change in scalability requirements as an example. We have verified the hypothesis that flexibility creates real options in the structure relative to the likely change. We have shown how the uncertainty, attributed to the likelihood of the change, makes real options theory superior to other valuation techniques which fall short in dealing with the value of architectural flexibility under uncertainty. We have compared the options results to other valuation techniques, PV and DCF, where the latter fall short in dealing with the value of architectural

flexibility under uncertainty. The results show that ArchOptions yield more realistic measures under uncertainty, as it continue to account for the embedded options in a system of a given architecture. We have exemplified and demonstrated the comprehensiveness and the effectiveness of the valuation points of view framework in "capturing" the options on an evolving architecture from two valuation points of view. This is necessary for reaching a comprehensive value of options from different perspectives. We have verified the claim that the decision of selecting a potentially stable architecture has to maximize the value added relative to some valuation points of view. For this case, we have particularly shown that the choice of a "more" stable distributed software architecture has to be guided by the choice of the underlying middleware and its flexibility in responding to future changes in scalability requirements and relative to two valuation points of view. These are the maintainability and the throughput valuation points of view. The overall results show that value-based reasoning and real options can provide better insights on stability and investment decisions related to the evolution of software architectures.

On the discipline level, the application of ArchOptions to the above cases has drawn some preliminary observations, lessons, and insights that could stimulate future research in the area of relating requirements to software architectures. Consequently, these observations advance our understanding to the architectural stability problem, when addressed from an evolution and economics-driven software engineering perspectives. For example, the case of the middleware-induced architectures provides the reader with a fair amount of insight into the complexity and economic ramifications of a typical critical change in non-functional requirements (i.e., changes in scalability) and its impact on the architecture. Note that in-depth analysis of the change in critical non-functionality like scalability, its impact on the architecture, and its economics implications are often ignored and left unaddressed in the requirements and the architectures research. This, we believe, is just a step towards a better understanding of how critical non-functional requirements could relate to the architecture and tend to evolve as the requirements evolve.

Though ongoing research on the "coupling" of middleware and architectures(e.g., [Jazayeri, 1995; Gall et al., 1997; Sullivan et al., 1997; Oreizy et al., 1998; Di Nitto and Rosenblum, 1999; Metha et al., 2000; Denaro et al., 2004]) could have an impact on

understanding the relation between architectures and non-functional requirements, their contributions to such understanding is still insufficient. As far as the architectural stability problem is concerned, no effort has been devoted for understanding the evolution of non-functional requirements in relation to both the architecture and the middleware, when coupled. Our use of architectural flexibility and its value as metric to inform the decision of selecting a "more" stable middleware-induced architecture is novel and only a step toward such an understanding using a value-based reasoning.

Researchers working on relating requirements to architectures (e.g., [Finkelstein, 2000; van Lamsweerde, 2000; Nuseibeh, 2001]) have often begged the question: which architectural styles tend to be more stable in face of likely changes in requirements? Our observations have reshaped this question. In particular, the results- of Section 6.3 - have shown that though two architectures have exhibited similar styles (i.e., three-tier styles), they have differed in the way they cope with likely changes in scalability requirements. The governing factor, hence, appears to be to a large extent dependent on the flexibility of the middleware (e.g., through its built-in primitives) in supporting the change. The intuition and the preliminary observations, therefore, suggest that the style alone is not enough for answering this question, as when the non-functional requirements evolve. Understanding architectural stability relative to changes in non-functional requirements is also a factor of the extent to which the middleware primitives can support changes in non-functional requirements. Though this is an interesting observation, its validity is subject to further careful empirical studies.

Chapter 7

Conclusions, Future Work, and Open Questions

In this chapter, we summarize the thesis contribution. We highlight some future work on ArchOptions. We conclude by highlighting some open questions that could stimulate future research in architectural stability, relating requirements to software architectures, and architectural economics.

7.1 Summary of the Contribution

The main goal of the thesis has been the development of a framework for systematically evaluating the stability of software architectures in face of changes in requirements, taking an economics-driven approach. The contribution could be summarized as follows:

We have reviewed research work on architecture evaluation and have discussed their limitations in addressing architectural evaluation for stability. We have investigated the requirements for evaluating architectural stability from an economics-driven software engineering perspective and have described a real options-based model to address these requirements. We have complemented the model with a three-phase method for conducting an architectural evaluation for stability. The method provides guidelines on eliciting the likely changes in requirements and relating architectural decisions to value. For valuing flexibility of

an architecture to change, the method includes a valuation points of view framework, which we have outlined. The framework accounts for the economic ramifications of the change on the structural (e.g., maintainability) and behavioral (e.g., throughput) qualities of an architecture and on relevant business goals (e.g., new market products). We have exemplified and demonstrated the comprehensiveness of the valuation points of view framework in capturing the options on an evolving architecture from different perspectives. This framework is viable for the decision of selecting a potentially stable architecture has to maximize the value added relative to some valuation points of view.

In evaluating the thesis in the large, we have explored the approach "fitness" in addressing two architectural-centric software evolution problems. These are (i) assessing the worthiness of reengineering for change, and (ii) informing the selection of a "more" stable middleware-induced software in the face of changes in non-functional requirements. Addressing these problems have resulted in novel applications of real options theory in valuing the payoff of refactoring [Bahsoon and Emmerich, 2004b] and in informing the selection of middleware-induced software architectures using options[Bahsoon et al., 2005]. In evaluating the thesis in the small, we have verified the claim that the flexibility of an architecture in face of likely changes has values in the form of real options. We have shown how the uncertainty, attributed to the likelihood of the change, makes real options theory superior to other valuation techniques which fall short in dealing with the value of architectural flexibility under uncertainty. We have compared the options results to other valuation techniques, PV and DCF, where the latter fall short in dealing with the value of architectural flexibility under uncertainty. The overall results show that our approach yields more realistic measures for the value of architectural flexibility under uncertainty, as the approach accounts for the embedded growth options in a system of a given architecture. We have used general qualitative characteristics including simplicity of use, openness, comprehensiveness, and prediction effectiveness to further evaluate the thesis.

On the discipline level, the application of ArchOptions to the above cases has drawn some preliminary observations, lessons, and insights that could have implications on future research in the area of relating requirements to software architectures.

7.2 Future Work on ArchOptions

Multi-objective optimization view to design and the interdependence of non-functional requirements

We have taken the view that software design and engineering activity is one of investing valuable resources under uncertainty with the goal of maximizing the value added [Sullivan, 1996]. It is possible to adopt a complex view of value. One could characterize software design as a multi-objective optimization activity in which one trades safety for performance, or in which one satisfies multiple stakeholders [Boehm, 1989]. We have taken a narrow view to valuation: the value is measured relative to one objective at a time. For example, upon applying the ArchOptions model to select a "more" stable middleware-induced software architectures, we have relaxed considering the change impact of scaling up the software system on other non-functional requirements like security, availability, and reliability to optimize for these interacting requirements. However, we note that the analysis might get complex upon accounting for the impact of the change on other non-functional requirements and their interactions. Note the change could positively or negatively affect other non-functional requirements. For the refactoring case, we have valued the payoff of investing in a refactoring exercise relative to future savings in maintainability. We, however, acknowledge the fact that refactoring could also have implications on other quality of the structure such as extensibility, modularity, reusability, or efficiency. If we take the multi-optimization view to software design, understanding the cost/value implications is not straightforward and worth a separate investigation. In this context, utilizing the NFR framework [Mylopoulos et al., 1992], for example, could be a promising starting point to model the interaction of various non-functional requirements, their corresponding architectural decisions, and the negative/positive contribution of the architectural decisions in satisfying these non-functionalities. The framework could be then complemented by means for measuring (i) the corresponding cost of implementing the change itself, and (ii) the additional cost due to the impact of the change on other contributing or conflicting non-functionalities.

Valuation of the architectural potential to the change

As we have acknowledged, the problem of valuing the architectural potential to the change is a multi-perspective valuation problem. In today's world of rapidly changing information technology, organizations, and marketplaces, the requirements tend also to change, and in ways that require participation of all knowledgeable parties to value the architectural potential to the change. This necessitates finding a comprehensive solution for capturing the value from different perspectives. In chapter 5, we have highlighted a framework for addressing this problem. The valuation point of view framework aims at providing a comprehensive solution for quantifying the options from different perspectives. Future work may entail finding ways to manage the valuation under this framework, such as identifying the dimensions, which are critical for understanding architectural stability, prioritizing and weighting the valuation of these dimensions, managing conflicts, and reconciling the options results. This is necessary to provide a sound comprehensive valuation, which takes into account the various valuation points of views. The model interpretations and decision-making may then need to be tuned accordingly. Though both contributions are unrelated and address different problems, it would be possible for future research on ArchOptions to benefit from existing work on viewpoints frameworks (e.g., [Nuseibeh et al., 1994]). This because the highlighted framework inherits and mimics much of the characteristics described in viewpoints frameworks (e.g. "modularity" and "separation of concerns"); it follows the trend towards heterogeneity in reasoning. Up to the author's knowledge, no work has been done on exploiting viewpoints in the economics-driven software engineering research. This will demonstrate the ability to leverage the contribution on robust approaches in software engineering to solve problems in an emerging discipline, the value-based software engineering.

Further application of the model: aspects and architectural economics

The success and popularity of aspect-oriented software development have created an interest in transforming existing software systems into aspect-oriented ones. Such a transformation tends to improve the value of the structure, through the separation of concerns, but incurs upfront costs. The upfront costs include the cost of identifying

potential aspects and the crosscutting concerns in existing non-aspect-oriented system; the cost of refactoring a non-aspect into an aspect-oriented one; and the cost of "evolving" the associated maintenance-related infrastructure as a result of such transformation (e.g., generating new test suites following the transformation). The benefits, if any, are due to the enhanced flexibility in the structure. These benefits, however, are uncertain, long-term, and may not be immediate. The benefits may take the form of expected savings in maintenance and/or returns due to the enhancement of some qualities such as maintainability, extensibility, modularity, reusability, or efficiency.

The problem of understanding the economics of transforming non-aspect systems into aspect-oriented ones is appealing to the use of real options theory in general and ArchOptions in specific. Building on ArchOptions may result in economics models, which aim to quantify the payoffs of transforming a system into aspects. These models may inform the decision of investment through a tradeoff between the up-front costs and the expected benefits as a result of such transformation. These models may need to be derived empirically from real life cases to answer questions like: when is it cost-effective to invest in an aspect-transformation exercise? How can we value the payoff due to such transformation prior to investing in such an exercise? How can we reason about this payoff in connection with changes in the structure and at correspondingly higher level of abstractions than code? The studies and the derived models are likely to have an impact on understanding the economics of aspect-transformation activities, may result, or motivate economics-driven approaches to aspects.

7.3 Open Questions

Though the software architecture, as a key designed artifact, is considered to be "the promising solution for easing and guiding software maintenance and evolution" [Jazayeri, 2002], rapid technological advances and industrial evidence are now showing that the architecture is creating its own maintenance, evolution, and economics problems. Part of the problem stems in (i) the rapid technological advancements where evolution is not limited to a specific domain but extends to

"horizontally" cover several domains, (ii) the current practice in engineering requirements, which ignore the above, (iii) and the improper management of the evolution of these requirements and across different design artifacts of the software system. In the subsequent sections, we highlight some open issues that future research may consider to address some architectural-centric software evolution problems. Addressing these questions may have a positive implication on understanding the architectural stability problem.

Coping with rapid technological advancements and changes in the application domain

Assume that a distributed e-shopping system architecture which relies on a fixed network needs to evolve to support new services, such as the provision of mobile e-shopping. Moving to mobility, the transition may not be straightforward: the original distributed system's architecture may not be respected, for mobility poses its own non-functional requirements for dynamicity that are not prevalent in traditional distributed setting [Capra, 2003]. Examples of these requirements include the need to react to frequent changes in the environment, such as change in location; resource availability; variability of network bandwidth; the support of different communication protocols; losses of connectivity when the host need to be moved; and so forth. These requirements may not be satisfied by the current fixed architecture, the built-in architectural caching mechanisms, and/or the underlying middleware. Replacement of the current architecture and/or its underlying middleware may be required.

The challenge is thus to cope with the co-evolution of both the architecture and the non-functional requirements as we change domains. This poses challenges in understanding the evolution trends of non-functional requirements; designing architectures, which are aware of how these requirements will change over the projected lifetime of the software system and tend to evolve through the different domains. From an economics perspective, such is necessary to reduce the future "switching cost", which could hinder the success of evolution. In this perspective, engineering requirements and designing architectures need to be treated as value-maximizing activities in which we can maximize the net benefits (or real options) by

minimizing the future "switching costs" while transiting across different domains. This necessitates amending the current practice of engineering requirements and brings a need for methods and techniques, which explicitly model the domain, the "vertical" evolution of the software system within the domain itself and how the domain is likely to change over the projected lifetime of the software system. Again, goal-oriented requirements engineering could be a promising starting point to "horizontally" capture the evolution across various domains and "vertically" across the domain itself. The problem of selecting an architecture, which tend to be stable as the "vertical" and the "horizontal" requirements evolve, become a multi-optimization design problem, where the selected architecture must maximize the value added relative to the "vertical" and the "horizontal" changes. The modeling could be then complemented by valuation frameworks which have the promise for answering questions of interest such as which architectural styles and middlewares, have the promise to reduce the switching costs and could prevail over the life time of the software system? This we believe is a practical need for engineering requirements to support stable software architectures.

Architectural stability: the architecture or the middleware?

Recent research effort (e.g., [Jazayeri, 1995; Gall et al., 1997; Sullivan et al., 1997; Oreizy et al., 1998; Di Nitto and Rosenblum, 1999; Metha et al., 2000; Denaro et al., 2004]) on the relation between software architectures and middleware has been motivated by pragmatic needs. The effort has revolved on issues such as investigating the compliancy of architectural styles with middleware; capabilities that the middleware and the architecture can bring when "coupled" to understand quality attributes of the system such as performance; mapping between middleware and software architectures; and semantics and syntactical issues related to the mapping process. As it has been noted in several occasions [Emmerich 2000b; Emmerich 2002], research on software architectures has over-emphasized functionality and not sufficiently addressed how global properties and non-functional requirements are achieved in an architecture, where these requirements cannot be attributed to individual components or connectors. Though we believe that ongoing research on the "coupling" of middleware and architectures could have an impact on understanding the relation between architectures and non-functional

requirements, their contributions to such understanding is still insufficient. As far as the architectural stability problem is concerned, no effort has been devoted for understanding the evolution of non-functional requirements in relation to both the architecture and the middleware, when coupled. Our use of architectural flexibility and its value as metric to inform the decision of selecting a "more" stable middleware-induced architecture is novel but only a step toward such an understanding using a value-based reasoning. Some of the results are still preliminary: though, for example, the two middleware-induced architecture have exhibited similar three-tier styles, these architectures have differed in the way they cope with the change in scalability. Our preliminary observations suggest that the style by itself is not revealing to the analysis of architectural stability with respect to changes in non-functional requirements. Though this observation reveals a trend that agrees with the intuition and the state-of-practice, confirming the validity of these observations are still subject to some systematic empirical studies. These studies may need to consider other non-functional requirements, their concurrent evolution, and their corresponding change impact on different architectural styles and middleware, which worth future research.

Change management: traceability of requirements to the architecture

An important outcome of the initial development of the software system is the knowledge that the development team acquires: the knowledge of the application domain, user requirements, role of the application in the business process, solutions and algorithms, data formats, strength and weakness of the architecture, and operating environment. This knowledge is acknowledged to be crucial prerequisite for evolution [Bennet and Rajlich, 2000]. In particular, both the architectures and the team knowledge make the evolution possible [Bennet and Rajlich, 2000]. These to a great extent allow the team to make changes in the software without damaging the architectural integrity. Once one or the other aspect disappears, the system is no longer evolvable and enters the stage of servicing (also referred to as maturity by Lehman) [Bennet and Rajlich, 2000]. At the servicing stage, only small tactical changes would be possible. For the business, the software is likely to be no longer a core product and the cost-benefit of the change becomes marginal. According to Bennet and Rajlich [2000], there is a positive feedback between the loss of software

architecture coherence and the loss of software knowledge. Less coherent architectures requires more extensive knowledge in order to evolve the system of the given architecture. However, if the knowledge necessary for evolution is lost, the changes in the software will lead to faster deterioration of the architecture. Very often on software projects, the loss of knowledge is triggered by loss in key personnel and the project slips into the servicing stage. Hence, planning for evolution and stable software architectures urges the need for traceability techniques, which traces requirements and their evolution back and forth into the architecture and aid in "preserving" the team knowledge.

Davis [1993] gives the earliest definition of traceability. Davis defines traceability as "the ability to describe and follow (track) the lifetime of an artifact, in both a forward and a backward direction, i.e., from its origin to development and vice versa" [Davis, 1993]. Gotel and Finkelstein [1995] have preserved the spirit of Davis's definition of traceability. They, however, have scoped the definition on tracing a requirement through its "life". The requirements life covers periods of a requirement origin, development and specification, deployment, use, and on-going refinement. They have defined *requirements traceability* as "the ability to describe and follow the life of a requirement in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases)". Gotel and Finkelstein [1995] have particulary discussed the importance of tracing requirements back to their source. These sources might be people, other requirements, documents, or standards.

Traceability is important for modeling dependencies among software objects and for managing the change across software artifacts. Traceability information records the dependencies between requirements and the sources of these requirements, dependencies between requirements themselves, and dependencies between requirements and the system implementation [Kotonya and Sommerville, 1998]. Advances in software-development environments and repository technology have enabled software engineers to trace the change in software using traceability techniques. According to [Gotel and Finkelstein, 1995], these techniques span a variety of approaches ranging from cross-referencing schemes (e.g., cross-referencing

schemes, based on some form of tagging, numbering, indexing, traceability matrices, and matrix sequences), through document-centered techniques (e.g., Templates, hypertext, and integration documents), to more elaborate structure-centered techniques (e.g., assumption-based truth maintenance networks, constraint networks, axiomatic, key phrase, and/or relational dependencies).

We define *requirement to architecture traceability* as the ability to describe the "life" of a requirement through the requirements engineering phase to the architecture phase in both forwards and backwards. Forwards demonstrates which (and how) architectural element(s) satisfy an individual requirement in the requirements specification. Backwards demonstrates which requirement(s) in the requirements specification an individual architectural element relate to and satisfy. Current architectural practices, however, do not provide a support for traceability from the requirements specification to the architectural description (i.e., which and (how) requirement(s) in the requirements specification an individual architectural element relate to and satisfy and vise versa). Maintaining traceability "links" is necessary for managing the change, the co-evolution of both the requirements and the architecture, confining the change, understanding the change impact on both the structure and the other requirements, providing a support for automated reasoning about a change at a high level of abstraction. Further, such traceability "links" make it easier to preserve the acquired knowledge of the team through guided documentation. This may then minimize the impact of personnel losses, and may allow the enterprise to make changes in the software system without damaging the architectural integrity and making the software system unevolvable.

Architectural change impact analysis

Although change impact analysis techniques are widely used at lower levels of abstractions (e.g., code levels) and on a relatively abstract levels (e.g., classes in O.O. paradigms), little effort has been done on the architectural level (i.e., architectural impact analysis). Formal notations for representing and analyzing architectural designs generically referred to as Architectural Description Languages (ADLs) have provided new opportunities for architectural analyses [Garlan 2000]. Examples of such analyses includes system consistency checking [Allen and Garlan, 1994;

Luckham *et al.*, 1995], and conformance to constraints imposed by an architectural style [Abowd *et al.*, 1993].

Notable effort using dependency analysis on the architectural level includes the "chaining" technique suggested by Stafford, Richardson, and Wolf [1997]. The technique is analogous in concept and application to program slicing. In chaining, dependence relationships that exist in an architectural specification are referred to as links. Links connect elements of the specification that are directly related. The links produce a chain of dependencies that can be followed during analysis. The technique focuses the analysis on components and their interconnections. A component may have a set of input and output ports (which correspond to the component's interface). These ports may have been connected to one another to form a particular architectural configuration. Communication between components is accomplished by sending events to the component's ports. Stafford *et al.* [1997] supports the approach with an analysis tool, Aladdin. Aladdin accepts an architectural specification as input. A variety of computations can be then performed. The computations include unconnected component identification, change impact analysis (i.e., which components will be affected by an architectural change), and event dependence analysis (i.e., which components can send the following event to this port). These computations start at a particular component and/or port. Forward and/or backward chaining are then performed to discover related components. Forward and backward chaining is analogous in concept to forward and backward walk in the data-flow slicing. The applicability of this technique is demonstrated on small scale architectures and could be extended to address current architectural development paradigms. For example, how such a concept could be refined to perform what-if analysis on large-scale software architectures such as product-line or model-driven architectures? For product-line architectures, this is necessary for reasoning about how the change could impact the commonality, variability, and their interdependence. These techniques could be then complemented by analysis tools which could facilitate automated reasoning and provide a basis for what-if analyses to manage the change across instances of the core architecture. Understanding how the change could then ripple across different products might be feasible. For model-driven architectures, for example, this could help in reasoning about how the change

could affect the Platform Independent Model (PIM) and ripple to affect the Platform Specific Models (PSM). These techniques could be complemented by automated reasoning to manage evolution. When combined with traceability links, the combination could provide a comprehensive framework for managing the change and guiding evolution.

Empirical studies

A key benefit of adopting an architecture-centric approach to manage the evolution of the software system is driven by the objective of reducing future evolution costs, while attaining a net benefit and embedding real options. Though this is the motivation behind many architectural-centric approaches to software evolution, such as product-line architectures and model-driven architectures, little -if no- documented empirical evidence is available on the extent to which the architecture has succeeded or failed in attaining its objectives. In particular, the architectural stability problem is just a hint on the fact that the architecture is also creating its own problems. This brings a need for systematic empirical studies to analyze real life horror cases, which lead to substantial "break" in the architecture of the software system upon accommodating changes in requirements. The "breakage" could be attributed to the nature of the change, personnel, the architectural style, the adopted middleware, and so forth. Lessons to be learned from these studies may have positive implications on the way we engineer our future requirements, design architectures to meet these changing requirements, and have better understanding on how we can control risks associated with the change and its impact. The main objective is to learn from the state-of-practice to improve the state-of-the-art.

Concluding remarks

The thesis is a culmination of four years of independent "make a way" challenge into the architectural stability problem, in the absence of closely related focused research. The contribution may have the following implications: advancing the understanding of the architectural stability and its related problems from an economics-driven perspective, stimulating, and possibly motivating future research in architectural stability and related problems. The intellectual framework is most critical; the thesis

demonstrates that with *value-based* reasoning we can improve our ability to evaluate for architectural stability, develop software systems that need to adapt to the inevitable evolving requirement, and provide a basis for analyzing the stability and investment decisions for many architecture-centric evolution problems.

Appendix A

The COnstructive COst MOdel (COCOMO): Brief Background

The COCOMO (COnstructive COst MOdel) cost and schedule estimation model was originally published in [Boehm 1981]. It became one of most popular parametric cost estimation models of the 1980s. However, COCOMO '81 along with its 1987 Ada update experienced difficulties in estimating the costs of software developed to new life-cycle processes and capabilities. Boehm validated his COCOMO model in the 1980's and he obtained very good results for the intermediate and detailed COCOMO, and quite poor ones for the basic COCOMO. Independent evaluations performed on other data sets have not always produced such good results, with results fluctuating from high to low accuracy in predictions. For example, it was found that COCOMO I may systematically overestimate the effort. COCOMO I has been improved and resulted in the so called COCOMO II. COCOMO II improves the estimation by incorporating expert knowledge using Bayesian Statistics. Such a calibration has lead COCOMO II to reach promising results outperforming COCOMO I.

In particular, the COCOMO II research effort was started in 1994 at USC to address the issues on non-sequential and rapid development process models, reengineering, reuse driven approaches, object oriented approaches, etc. COCOMO II [Boehm et al., 1995] has three submodels, Applications Composition, Early Design and Post-Architecture, which can be combined in various ways to deal with the current and likely future software practices marketplace. The Application Composition model is used to estimate effort and schedule on projects that use Integrated Computer Aided

Software Engineering tools for rapid application development. These projects are too diversified but sufficiently simple to be rapidly composed from interoperable components. Typical components are GUI builders, database or objects managers, middleware for distributed processing or transaction processing, etc. and domain-specific components such as financial, medical or industrial process control packages. The Applications Composition model is based on Object Points [Banker et al., 1994; Kauffman and Kumar, 1993]. Object Points are a count of the screens, reports and 3GL language modules developed in the application. Each count is weighted by a three-level; simple, medium, difficult; complexity factor. This estimating approach is commensurate with the level of information available during the planning stages of Application Composition projects. The Early Design model involves the exploration of alternative system architectures and concepts of operation. Typically, not enough is known to make a detailed fine-grain estimate. This model is based on function points (or lines of code when available) and a set of five scale factors and seven effort multipliers. The Post-Architecture model is used when top level design is complete and detailed information about the project is available and as the name suggests, the software architecture is well defined and established. It estimates for the entire development life-cycle and is a detailed extension of the Early-Design model. This model is the closest in structure and formulation to the Intermediate COCOMO '81 and Ada COCOMO models. It uses Source Lines of Code and/or Function Points for the sizing parameter, adjusted for reuse and breakage; a set of 17 effort multipliers and a set of 5 scale factors, that determine the economies/diseconomies of scale of the software under development. The 5 scale factors replace the development modes in the COCOMO '81 model and refine the exponent in the Ada COCOMO model. The Post-Architecture Model has been calibrated to a database of 161 projects collected from Commercial, Aerospace, Government and non-profit organizations using the Bayesian approach [Chulani *et al.*, 1998]. The Early Design Model calibration is obtained by aggregating the calibrated Effort Multipliers of the Post-Architecture Model as described in [USC-CSE, 1997]. The Scale Factor calibration is the same in both the models. Unfortunately, due to lack of data, the Application Composition model has not yet been calibrated beyond an initial calibration to the [Kauffman and Kumar, 1993] data. A primary attraction of the COCOMO models is their fully-available internal equations and parameter values. Over a dozen commercial COCOMO '81

implementations are available; one (Costar) also supports COCOMO II: for details, see the COCOMO II website http://sunset.usc.edu/COCOMOII/suite.html.

Appendix B

Further Supporting Material: The Middleware-Induced Architecture Case

In this appendix, we provide supporting material for the case of using ArchOptions to select a "more" stable middleware-induced architecture, described in Section 6.3 of Chapter 6.

B.1 Description of the fault tolerance architecture

We describe the components of the Fault Tolerance Infrastructure as shown on the top of Figure 6.5 of Chapter 6. These include *Replication Manager*, *Fault Notifier*, and *Fault Detector*. The bottom of Figure 6.5 shows three hosts: H_1, H_2, and H_3. The client application object C on H_1 invokes a replicated server object with two replicas S_1 on host H_2, and S_2 on host H_3. The Figure shows Factory and Fault Detector objects that may be present and specific for a host. The service objects are replicated objects. The host-specific objects, however, are not replicated. The figure also shows the Message Handler and the Logging and Recovery Mechanisms that are present on each host. Logically, a single instance of the Replication Manager and Fault Notifier shall exist in each fault tolerance domain. Physically, however, they are replicated to protect against faults, as any other application object are.

B.2 Description of the load balancing architecture

Figure 6.6 of Chapter 6 illustrates the components in TAO's load balancing service. The design supports adaptive load balancing and on-demand request forwarding [Othman et al. 2001b] and outlined below:

The *Replica Locator* identifies which of the replicas will be assigned a request. The *Replica Locator* component forwards the requests to the *Load Analyzer* component. The *Load Analyzer* component analyses the requests; it select the replica to be assigned the request. The *Replica Locator* obtains a reference to a replica from the load analyzer and then forwards the request to that replica. The *Replica Locator* binds clients to the identified replicas. The *Load Analyzer* also allows explicit selection of a load balancing strategy at runtime, while maintaining a simple and flexible design. The replica locator is portably implemented using servant locators implementing the interceptor pattern [Schmidt et. al., 2000], abiding to standard CORBA portable object adapter mechanisms [Henning and Vinoski, 1999]. The *Load Balancer* component is a mediator that integrates all the components. It provides an interface for load balancing without exposing clients to the intricate interactions between the components it integrates. The *Load Monitor* component monitors loads on a given replica, reports replica load to a *Load Balancer*, and informs replicas when they should accept requests versus forward them back to the load balancer. Each object that TAO's load balancing service manages communicates with it through a unique proxy. The load balancer uses the *replica proxies* components to distinguish different replicas to workaround CORBA's so-called "weak" notion of object identity [Object Management Group, 1999], where two references to the same object might have different values.

B.3 Implementation of the fault tolerant, the load balancing Services, and their Change Impact on the CORBA-induced architecture

A List of classes and files necessary to implement the fault tolerant service into the Duke's Bank architecture of Section 6.3 of Chapter 6 is depicted in Table B-1. Table B-2 reports on the effort necessary to develop and integrate the load balancing service into the middleware.

Table B-1. Implementing the fault tolerance service on CORBA

CosFaultTolerance	IDL	242	Interface description of remote methods
PropertyManagerImpl	Java	273	Implementation of the PropertyManager interface
ObjectGroupManagerImpl	Java	672	Implementation of the ObjectGroupManager interface
GenericFactoryImpl	Java	523	Implementation of the GenericFactory interface
ReplicationManagerImpl	Java	865	Implementation of the ReplicationManager interface
FaultNotifier	Java	611	Implementation of the FaultNotifier interface
ClientPolicy	Java	155	Implementations of the RequestDurationPolicy interface
ServerPolicy	Java	61	Implementation of the HeartbeatEnabledPolicy
FTPolicy	Java	207	Implementation of the HeartbeatPolicy interface
FaultDetector	Java	149	Class defining the component illustrated above
DefaultFaultAnalyzer	Java	113	The default fault analyzer
ReplicationManagerFaultAnalyzer	Java	865	Replication Manager's fault analyzer
FaultConsumer	Java	200	Connect to the fault notifier
PropertyValidator	Java	29	Class providing static methods to validate properties
MemberInfo	Java	50	Structure that contains all member-specific information
PropertyUtils	Java	53	Provides some methods used to manipulate properties
Operators	Java	23	Class providing static methods related to operators
ReplicationManagerServer	Java	13	Class running the Replication Manager server
FaultNotifierServer	Java	13	Class running the Fault Notifier server
Total		5117	

Table B-2. Implementing the load balancing service on CORBA

File Name	File Type	SLOC	Description
CosLoadBalancing	IDL	90	Interface description of remote methods
LoadAlertImpl	Java	26	Implementation of LoadAlert interface.
LoadCPUMonitorImpl	Java	138	LoadMonitor implementation that monitors the overall CPU load on a given host
LoadManagerImpl	Java	919	Implementation of LoadManager interface
LeastLoaded	Java	405	Implementations of Strategy interface
LoadAverage	Java	305	Implementations of Strategy interface
LoadMinimum	Java	389	Implementations of Strategy interface
RoundRobin	Java	121	Implementations of Strategy interface
Random	Java	128	Implementations of Strategy interface
MemberLocator	Java	59	Class which defines the component described above
LoadAlertHandler	Java	40	This class handles all asynchronously received replies from all registered LoadAlert objects. It only exists to receive asynchronously sent exceptions
LoadAlertInfo	Java	30	Structure that contains all LoadAlert-specific information
LoadAlertMap	Java	60	Maps a LoadAlertInfo with a location
LoadListMap	Java	60	Maps a LoadList with a location
LoadMap	Java	60	Maps a load with a location
MonitorMap	Java	60	Maps a LoadMonitor with a location
PullHandler	Java	58	Event handler used when the "pull" monitoring style configured
PushHandler	Java	39	Event handler used when the "push" monitoring style is configured
LB_ServerRequestInterceptor	Java	109	Responsible for redirecting the requests back to the manager
LB_ORBInitializer	Java	72	*Creates and registers with the ORB the* LB_IORInterceptor and LB_ServerRequestInterceptor
LB_ClientRequestInterceptor	Java	62	Handles transparent object group member registration with the LoadManager, and registration of the LoadAlert object necessary for load shedding
LB_ClientORBInitializer	Java	33	Creates and registers with the ORB the LB_ClientRequestInterceptor
LoadManagerServer	Java	214	Class running the load balancer server
LoadMonitorServer	Java	315	Class running the load monitor server
Total		3943	

Appendix C

Discount Cash Flows (DCF) and Net Present Value (NPV): Brief Explanation

According to [Trigeorgis, 1995] in finance, the cost and benefits associated with an investment are called cash flows. Investments are compared only on the basis of cash flows. Usually, there is an original investment, C_o, represented as a negative number. SubseTuent cash Ilows alé denoted as Cash Flow YeaU1,..., Cash Flow YeaUn, spanning the time horizon in which the investment incurs costs and generates benefits. The Present s alue (Ps Fof a future cash flow is the value of the cash flow as though it was received toady.

Moving forward from present to future, an investment is expected to grow at a certain rate of return. Now turning it around: Moving backward from future to present, an investment shrinks with the same rate of return. t hen moving back in time, the rate of backward adjustment is itself is called discounting. The general technique of valuing a capital investment project by summing its discounted future cash flows is known as discounted cash flows (a CcF. Simply the a Cc is obtained by the discounted benefits minus the discounted costs as given in the below formula:

$$DCF = \frac{\text{Cash Flow Year 1}}{(1+r)^1} + \frac{\text{Cash Flow Year 2}}{(1+r)^2} + \frac{\text{Cash Flow Year n}}{(1+r)^n}$$

The Cash flow Year i, represents cash flows in which the cash flows occur, and r is a per-period discount rate. The formula simply tells that whether an investment is worth more than it costs. The rule is that if DCF is positive, the investment is worth undertaking; that is, it generates more value than it costs. If DCF is negative, it should be forgone as the investment generates less value than its costs. If it is zero, we are indifferent between undertaking and forgoing it.

Glossary of Economics Terms

American option	*An option that may be exercised at any time up to and including the expiration date*
Call Option	*An option contract that gives its holder the right (but not the obligation) to purchase a specified number of shares of the underlying stock at the given strike price, on or before the expiration date of the contract.*
Cash flow	*In investments, cash flow represents earnings before depreciation, amortization, and non-cash charges. Sometimes called cash earnings. Cash flow from operations (called funds from operations by real estate and other investment trusts) is important because it indicates the ability to pay dividends*
Discount Cash Flows	*Future cash flows multiplied by discount factors to obtain present values*
European option	*Option that may be exercised only at the expiration date.*
Exercise price	*The price at which the security underlying a options contract may be bought or sold*
In-the-money	*A call option with a strike price lower than the underlying futures price. For example, if the March COMEX silver futures contract is trading at $6 an ounce, a March call with a strike price of $5.50 would be considered in the money by $0.50 an ounce.*
Net present value (NPV)	*The present value of the expected future cash flows minus the cost.*
Option	*Gives the buyer the right, but not the obligation, to buy or sell an asset at a set price on or before a given date. Investors, not companies, issue options. Buyers of call options bet that a stock will be worth more than the price set by the option (the strike price), plus the price*

they pay for the option itself.

Out-of-the-money *A call option is out of the money if the strike price is greater than the market price of the underlying security. That is, you have the right to purchase a security at a price higher than the market price, which is not valuable.*

Rate of return *The ratio of the additional annual income or profit generated by an investment to the cost of the investment. Here's a simple example, although the calculations are usually a great deal more involved for actual investments. If the cost of constructing a new factory is $10 million and it gives you an extra $1 million in profit each year, then its rate of return is 10 percent.*

Strike price *The stated price per share for which underlying stock may be purchased (in the case of a call) or sold (in the case of a put) by the option holder upon exercise of the option contract.*

Bibliography

[Abowd et al., 1993] Abowd, G., Allen, R., and Garlan, D.: Using Style to Understand Descriptions of Software Architecture. In: Proceedings of Foundations of Software Engineering, ACM Press (1993) 9-20

[Abowd et al., 1996] Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., and Zaremski, A.: Recommended Best Industrial Practice for Software Architecture Evaluation (CMU/SEI-96-TR-025), Software Engineering Institute, Carnegie Mellon University (1996)

[Allen and Garlan, 1994] Allen, R., and Garlan, D.: Formalizing Architectural Connection. In: Proceedings of the 14th International Conference on Software Engineering, ACM Press (1994) 71-80

[Amram and Kulatilaka, 1999] Amram, M., and Kulatilaka, N.: Real Options: Managing Strategic Investment in an Uncertain World. Harvard Business School Press, Cambridge, Massachusetts (1999)

[Antón, 1996] Antón, A.I.: Goal-based Requirements Analysis. In: Proceeding of the 2nd IEEE International Conference on Requirements Engineering, IEEE CS Press (1996) 136-144

[Antón, 1997] Antón, A.I.: Goal Identification and Refinement in the Specification of Software-Based Information Systems, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, GA (1997)

[Asundi and Kazman, 2001] Asundi, J. and Kazman, R.: A Foundation for the Economic Analysis of Software Architectures. In: Proceedings of the Third Workshop on Economics-Driven Software Engineering Research (2001)

[Bahsoon and Emmerich, 2003a] Bahsoon, R. and Emmerich, W.: Evaluating Software Architectures: Development, Stability, and Evolution. In: Proceedings of IEEE/ACS Computer Systems and Applications, IEEE CS Press (2003a) 47-57

[Bahsoon and Emmerich, 2003b] Bahsoon, R. and Emmerich, W.: ArchOptions: A Real Options-Based Model for Predicting the Stability of Software Architecture. In: Proceedings of the Fifth Workshop on Economics-Driven Software Engineering Research, in Conjunction with the 25th International Conference on Software Engineering, Portland, USA, IEEE CS (2003b) 35-40

[Bahsoon and Emmerich, 2004a] Bahsoon, R. and Emmerich, W.: Evaluating Architectural Stability with Real Options Theory. In: Proceedings of the 20th IEEE Int. Conference on Software Maintenance, Chicago, Illinois, IEEE CS Press (2004a) 443-447

[Bahsoon and Emmerich, 2004b] Bahsoon, R. and Emmerich, W.: Applying ArchOptions to Value the Payoff of Refactoring. In: Proceedings of the Sixth Workshop on Economics-Driven Software Engineering Research, in Conjunction with the 26th International Conference on Software Engineering, IEE Press (2004b) 66-70

[Bahsoon and Emmerich, 2005] Bahsoon, R. and Emmerich, W.: Using ArchOptions to Value the Flexibility of Product-Line Architectures. UCL Dept. of Computer Science, Research Note (2005)

[Bahsoon et al., 2005] Bahsoon, R., Emmerich, W., and Macke, J.: Using ArchOptions to Select Stable Middleware-Induced Architectures. In: IEE Proceedings Software, Special issue on Relating Requirements to Architectures, IEE Press 152(4) (2005) 176-186

[Bahsoon, 2003] Bahsoon, R.: Evaluating Software Architectures for Stability: A Real Options Approach. In: Proceedings of the Doctoral Symposium of the 25th International Conference on Software Engineering, IEEE CS Press (2003)

[Baldwin and Clark, 1993] Baldwin, C. Y., and Clark, K.B.: Modularity and Real Options. Working paper, Harvard Business School (1993)

[Baldwin and Clark, 1997] Baldwin, C. Y., and Clark, K.B.: Managing in the Age of Modularity. Harvard Business Review, 75 (5) (1997) 84-93

[Baldwin and Clark, 2001] Baldwin, C. Y., and Clark, K.B.: Design Rules - The Power of Modularity. MIT Press (2001)

[Belady and Lehman, 1976] Belady, L.A., and Lehman, M.M.: A Model of Large Program Development. IBM Systems Journal, 15(3) (1976) 225-252

[Bennet and Rajilich, 2000] Bennet, K. and Rajilich, V.: Software Maintenance and Evolution: A Roadmap. In: A. Finkelstein (ed.): The Future of Software Engineering. ACM Press (2000) 73-90

[Bergey et al., 2001] Bergey J., O'Brien, L., Smith, D.: Options Analysis for Reengineering (OAR): A Method for Mining Legacy Assets, CMU/SEI-2001-TN-013,ADA395201 (2001)

[Black and Scholes, 1973] Black, F., and Scholes, M.: The Pricing of Options and Corporate Liabilities. Journal of Political Economy. U. of Chicago Press (1973) 637-654

[Boehm and Ross, 1989] Boehm, B.W., and Ross, R.: Theory-W Software Project Management: Principles and Examples. IEEE Transactions on Software Engineering, 15(7) (1989) 902-916

[Boehm and Sullivan, 2000] Boehm, B., and Sullivan, K. J.: Software Economics: A Roadmap. In: A. Finkelstein (ed.): The Future of Software Engineering. ACM Press (2000) 320-343

[Boehm et al., 1995] Boehm, B., Clark, B., Horowitz, E., Madachy, R., Shelby, R., Westland, C.: The COCOMO 2.0 Software Cost Estimation Model. In: International Society of Parametric Analysts (1995)

[Briand and Wieczorek, 2002] Briand, I. and Wieczorek, L.: Resource Modeling in Software Engineering, Second edition of the Encyclopedia of Software Engineering, Wiley (2002)

[Brinkkemper, et al., 1996] Brinkkemper J., Lyytinen K., and Welke R. J.: Method Engineering. Principles of Method Construction and Tool Support. Chapman & Hall (1996)

[Burch et al., 1990] Burch, J., Clarke, E., McMillan, E., Dill, D., and Hwang, L.: Symbolic Model Checking: 10^{20} States and Beyond. In: Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science. IEEE CS (1990) 428-439

[Capra, 2003] Capra, L.: Reflective Mobile Middleware for Context-Aware Applications. PhD Thesis. University of London, UK (2003)

[Clements and Northrop, 2002] Clements, P., and Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley, Boston, USA (2002)

[Clements et al., 2002] Clements, P., Kazman, R., and Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison Wesley, Boston, USA (2002)

[Clements, 2000] Clements, P.: Active Reviews for Intermediate Designs. Technical Report (CMU/SEI-2000-TN-009), Software Engineering Institute, Carnegie Mellon University (2000)

[Coleman et al., 1994] Coleman, D., Arnold, P., Bdoff, S., Gilchrist, H., Hayes, F. and Jeremaes P., Object-Oriented Development: The Fusion Method. Prentice Hall (1994)

[Conte et al., 1986] Conte, S.D., Dunsmore, H.E., and Shen, V.Y: Software Engineering Metrics and Models. Menlo Park, Calif.: Benjamin-Cummings (1986).

[Cook et al., 2000] Cook S., Ji H. and Harrison R.: Software Evolution and Software Evolvability, working paper, U. of Reading, Aug. (2000)

[Cook et al., 2001] Cook, S., Ji, H., and Harrison, R.: Dynamic and Static Views of Software Evolution. In: International Conference on Software Maintenance, Florence, Italy. IEEE CS (2001) 592-601

[Corbett and Ayrunin, 1997] Corbett, J., and Avrunin, G.: Using Integer Programming to Verify General Safety and Liveness Properties. Formal Methods in System design, 6(2) (1997) 97-123.

[Cox and Rubinstein, 1979] Cox, J., Ross, S., and Rubinstein, M.: Option Pricing: A Simplified Approach. Journal of Financial Economics, 7 (3) (1979) 229-264

[Dardenne et al., 1993] Dardenne, A., van Lamsweerde A., and Fickas, S.: Goal-Directed Requirements Acquisition, Science of Computer Programming, 20(1-2) (1993) 3-50

[Davis, 1993] Davis, A: Software Requirements: Objects, Functions and States. Englewood Cliffs, New Jersey: Prentice-Hall (1993)

[Dawson et al., 2003] Dawson, R., Bones, P., Oates, B., Brereton, P., Azuma, M., and Jackson, M.: Empirical Methodologies in Software Engineering. In Eleventh Annual International Workshop on Software Technology and Engineering Practice, IEEE CS Press(2003) 52-58

[Denaro et al., 2004] Denaro, G., Polini A., and Emmerich W.: Performance Testing of Distributed Component Architectures. In: S. Beydeda and V. Gruhn (eds.), Building Quality into COTS Components - Testing and Debugging, Springer (2004) 294-314

[Di Nitto and Rosenblum, 1999] Di Nitto, E., and Rosenblum, D.: Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In: Proceedings of the 21st International Conference on Software Engineering, ACM Press (1999) 13-22

[Dixit and Pindyck, 1994] Dixit, A. and R. Pindyck: Investment under Uncertainty, Princeton University Press (1994)

[Dwyer and Clarke, 1994] Dwyer, M. and Clarke, L.: Dataflow Analysis for Verifying Properties of Concurrent Programs. In: Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Press (1994) 62-75

[EDSER 1-7, 1999-2005] EDSER 1-7: Proceedings of the Workshops on Economics-Driven Software Engineering Research: In conjunction with the 21st through 27th International Conference on Software Engineering (1999 - 2005)

[Emmerich, 2000a] Emmerich, W.: Engineering Distributed Objects. John Wiley & Sons, Chichester, UK (2000a)

[Emmerich, 2000b] Emmerich, W.: Software Engineering and Middleware: A Road Map. In: A. Finkelstein (ed.), Future of Software Engineering, ACM Press (2000b) 117-129

[Emmerich, 2002] Emmerich, W.: Distributed Component Technologies and their Software Engineering Implications. In: Proceedings of the 24th Int. Conference on Software Engineering, Orlando, Florida, ACM Press (2002) 537-546

[Erdogmus and Favaro, 2002] Erdogmus, H., and Favaro, J: Keep Your Options Open: Extreme Programming and Economics of Flexibility. In: XP Perspective, Addison Wesley (2002) 1-44

[Erdogmus and Vandergraaf, 1999] Erdogmus, H., and Vandergraaf. J: Quantitative Approaches for Assessing the Value of COTS-Centric Development. In: the Proceedings of the Sixth International Symposium on Software Metrics (METRICS' 99), Boca Raton, FL, IEEE CS Press (1999) 279-290

[Erdogmus et al., 2002] Erdogmus, H., Boehm, B., Harrison, W., Reifer, D. J., and Sullivan, K. J.: Software Engineering Economics: Background, Current Practices, and Future Directions.Tutorial Summary. In: Proceeding of 24th International Conference on Software Engineering, Orlando, FL, ACM Press (2002) 683-684

[Erdogmus, 2000] Erdogmus, H.: Value of Commercial Software Development under Technology Risk. The Financier 7(2000)

[FEAST1-2] Lehman, M.M.: Feedback, Evolution and Software Technology, FEAST 1-2. http://www-dse.doc.ic.ac.uk/~mml/feast/

[Finkelstein and Kramer, 2000] Finkelstein, A., and Kramer, J.: Future of Software Engineering. In: A. Finkelstein (ed.): The Future of Software Engineering, ACM Press (2000) 5-21

[Finkelstein, 2000] Finkelstein, A.: Architectural Stability. http://www.cs.ucl.ac.uk/staff/a.finkelstein/talks.html (2000)

[Formal Systems, 1992] Formal Systems (Europe) Ltd.: Failures Divergence Refinement: User Manual and Tutorial (1992)

[Gall et al., 1997] Gall, H., Jazayeri, M., Klösch, R., and Trausmuth, G.: The Architectural Style of Component Programming. COMPSAC, IEEE CS Press (1997) 18-27

[Gamma et al., 1995] Gamma E. et al.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Longman, Reading, Mass (1995)

[Garcia and Bray, 1997] Garcia, M.L. and Bray, O.H: Fundamentals of Technology Roadmapping, Sandia National Laboratories, www.sandia.gov/roadmap (1997)

[Garlan et al., 1994] Garlan, D., Allen, R., and Ockerbloom: Exploiting Style in Architectural Design Environments. In: Proceedings of SIGSOFT'94, Foundations of Software Engineering, New Orleans, Louisiana, USA, ACM Press(1994)175-188

[Garlan et al., 1995] Garlan, D., Monroe, R. and Wile, D.: ACME: An Architectural Interconnection Language. Technical Report (CMU-CS-95-219), Carnegie Mellon University (1995)

[Garlan, 2000] Garlan, D.: Software Architecture: A Roadmap. In: A. Finkelstein (ed.): The Future of Software Engineering, ACM Press (2000) 91-101

[Gilb, 1989] Tom, G.: Principles of Software Engineering Management, Addison-Wesley Longman (1989)

[Godefroid and Wolper, 1991] Godefroid, P., and Wolper, P.: Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In: Proceedings of the Third Workshop on Computer Aided Verification, Lecture Notes in Computer Sc, Springer (1991)417–428

[Gotel and Finkelstein, 1995] Gotel, O., and Finkelstein, A.: Contribution Structures. In: the Third Proceedings of the Requirements Engineering Symposium, York, UK, IEEE CS Press (1995) 169-178

[Henning and Vinoski, 1999] Henning, M., and Vinoski, S: Advanced CORBA Programming With C++, Addison-Wesley Longman, Reading, Mass (1999)

[Holzman, 1991] Holzman, G.: Design and Validation of Computer Protocol, Prentice Hall Software Series (1991)

[Hull, 1997] Hull, J. C.: Options, Futures, and Other Derivative Security. Third edition, Prentice-Hall (1997)

[IEEE Standard 610.12, 1993] IEEE Standard 610.12: Glossary of Software Engineering Terminology. In: Software Engineering Standards Collection, IEEE CS Press (1993)

[Jazayeri, 1995] Jazayeri, M.: Component Programming - a Fresh Look at Software Components, In: The Fifth European Software Engineering Conference. Lecture Notes in Computer Sc, Springer (1995) 457-478

[Jazayeri, 2002] Jazayeri, M.: On Architectural Stability and Evolution. Lecture Notes in Computer Science, Springer Verlag, Berlin (2002) 13-23

[JGroups] JGroups Website, http://www.jgroups.org.

[Jung, H.W., 1998] Jung, H.W. Optimizing Value and Cost in Requirements Analysis. IEEE Software (July/August) (1998) 74-78

[Karlsson and Ryan, 1997] Karlsson, J. and Ryan, K.: A Cost-Value Approach for Prioritizing Requirements. IEEE Software (September/October) (1997) 67-74

[Karlsson, et al., 1997] Karlsson, J., Olsson, S., and Ryan, K.: Improved Practical Support for Large-scale Requirements Prioritizing. Requirements Engineering Journal, 2(1) (1997)51-60

[Kataoka et al., 2002] Kataoka, Y., Imai, T., Andou, H., and Fukaya, T.: A Quantitative Evaluation of Maintainability Enhancement by Refactoring. In: Proceedings of the International Conference on Software Maintenance. IEEE CS (2002) 576-585

[Kazman et al., 1994] Kazman, R., Abowd, G., Bass, and L., Webb, M.: SAAM: A Method for Analyzing the Properties of Software Architectures. In: Proceedings of the 16th International Conference on Software Engineering, Sorento, Italy. IEEE CS (1994) 81-90

[Kazman et al., 1998] Kazman, R., Klein, M., Barbacci, M., Lipson, H., Longstaff, T., and Carrière, S.J.: The Architecture Tradeoff Analysis Method. In: Proceedings of fourth International Conference on Engineering of Complex Computer Systems (ICECCS '98), Monterey, CA, IEEE CS Press (1998) 68-78

[Kazman et al., 2001] Kazman, R., Asundi, J., and Klein, M.: Quantifying the Costs and Benefits of Architectural Decisions. In: Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, IEEE CS Press (2001) 297-306

[Kitchenham et al., 1997] Kitchenham, B., Linkman, S., and Law, D. : DESMET: A methodology for evaluating software engineering methods and tools. In IEE Computing and Control Journal (1997)

[Klein and Kazman, 1999] Klein, M., and Kazman, R.: Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-TR-22, Software Engineering Institute, Carnegie Mellon University (1999)

[Kotonya and Sommerville, 1998] G. Kotonya and I. Sommerville: Requirements Engineering: Processes and Techniques. John Wiley and Sons. May (1998)

[Kruchten, 2000] Kruchten, P.: The Rational Unified Process: An Introduction. Addison Wesley Longman (2000)

[Labourey and Burke, 2003] Labourey, S., and Burke B.: JBoss Clustering Documentation, JBoss Group LLC (2003)

[Lehman et al., 2000] Lehman, M.M., Kahen, G., and Ramil, J.F.: Replacement Decisions for Evolving Software. In: Proceedings of the Second Workshop on Economics-Driven Software Engineering Research (2000)

[Lehman, 1985] Lehman M. M.: Program Evolution. Academic Press, London (1985)

[Lehman, 1998] Lehman, M.M.: The Future of Software - Managing Evolution. IEEE Software (Jan. 1998)

[Leintz and Swanson, 1980] Leintz, B.P., and Swanson, E.B.: Software Maintenance Management. Addison-Wesley, Reading Mass (1980)

[Liu and Gorton, 2003] Liu, A. and Gorton, I.: Accelerating COTS Middleware Acquisition: The i-Mate Process. IEEE Software Vol. (20) (2) (2003) 72-79

[Luckham et al., 1995] Luckham, D. C., Augustin, L. Kenney, J., Vera, J, Bryan, M., and Mann W.: Specification Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, 21(4) (1995) 366-355

[Luckham, and Vera, 1995] Luckham, D.C., and Vera, J.: An Event-Based Architecture Definition Language. IEEE Transactions on Software Engineering, 29(9) (1995) 717-734

[Maciaszek and Liong, 2004] Maciaszek, L., and Liong, B.: Practical Software Engineering-A Case Study Approach. Addison-Wesley (2004)

[Madj and Pindyck, 1997] Madj, S. and R. Pindyck: Time to Build, Option Value, and Investment Decisions, Journal of Financial Economics, 18(1) (1997) 7-27

[Magee and Kramer, 1996] Magee, J., and Kramer, J.: Dynamic Structure in Software Architectures. In: Proceedings of the ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering, San Francisco, CA, ACM Press(1996) 3-14

[Magee et al., 1995] Magee, J., Dulay, D., Eisenbach, N., and Kramer, J.: Specifying Distributed Software Architecture. In: Proceedings of the Fifth European Software Engineering Conference (ESEC'95), Barcelona, Spain, Lecture Notes in Computer Sc, Springer (1995) 137-153

[Mansour and Bahsoon, 2002] Mansour, N. and Bahsoon, R. (2002): Reduction-based methods and metrics for selective regression testing, Journal of Information and Software Technology, Elsevier Science 40(7) (2002) 431-443,

[Masticola and Ryder, 1991] Masticola, S., and Ryder, B.: A Model of ADA Programs for Static Deadlock Detection in Polynomial Time. In: Proceedings of the Workshop on Parallel and Distributed Debugging (1991) 97-107

[Medvidovic and Taylor, 1997] Medvidovic, N., and Taylor, R.: A Framework for Classifying and Comparing Architecture Description Languages. In: Proceedings of the Sixth European Software Engineering Conference, together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, ACM Press (1997)60-76

[Medvidovic et al., 1999] Medvidovic, N., Rosenblum, D., and Taylor, R.: A Language and Environment for Architecture-Based Software Development and Evolution. In: Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA IEEE CS Press (1999)44-53

[Medvidovic et al., 2003] Medvidovic N., Dashofy E., and Taylor R.: On the Role of Middleware in Architecture-based Software Development. International Journal of Software Engineering and Knowledge Engineering, 13(4) (2003) 229-306

[Mehta et al., 2000] Mehta, N., Medvidovic, N., and Phadke, S.: Towards a Taxonomy of Software Connectors. In: Proceedings of the 22nd International Conference on Software Engineering, ACM Press (2000) 178-187

[Mens and Tourwe, 2004] Mens, T., and Tourwe, T.: A Survey of Software Refactoring. In: IEEE Transactions on Software Engineering, 30(2) (2004) 126-139

[Moricon et al., 1995] Moriconi, M., Qian, X., and Riemenschneider, R.: Correct Architecture Refinement. IEEE Transactions on Software Engineering, 21(4) (1995) 356-372

[Muller, 2002] Muller, G.: Roadmapping, Philips Research, http://www.extra.research.philips.com/natlab/sysarch/ (2002)

[Myers and Majd, 1990] Myers, S. and S. Majd: Abandonment Value and Project Life, Advances in Futures and Options Research, 4(1990) 1-21

[Myers, 1977] Myers, S.C.: Determinants of Corporate Borrowing. Journal of Financial Economics. 5(2) (1977) 147-175

[Myers, 1987] Myers, S. C.: Finance Theory and Financial Strategy. Corporate Finance Journal. 5(1) (1987) 6-13

[Mylopoulos et al., 1992] Mylopoulos, J., Chung, L., Nixon, B.: Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. IEEE Trans. on Software. Engineering, Vol. (18) (6) (1992)

[Naumovich et al., 1997] Naumovich, G., Avrunin, G.S, Clarke, L.A., and Osterweil, L.J.: Applying Static Analysis to Software Architectures. Technical Report, UM-CS-1997-008, University of Massachusetts, Amherst (1997)

[Nuseibeh and Easterbrook, 2000] Nuseibeh, B., and Easterbrook, S.: Requirements Engineering: A Roadmap. In: A. Finkelstein (ed.): The Future of Software Engineering, ACM Press (2000) 35-46

[Nuseibeh et al., 1994] Nuseibeh, B., Kramer, J., and Finkelstein, A.: A Framework for Expressing the Relationships between Multiple Views in Requirements Specification. Transactions on Software Engineering, 20(10) (1994)760-773

[Nuseibeh, 2001] Nuseibeh, B.: Weaving the Software Development Process between Requirements and Architectures. In: Proceedings of STRAW 01 the First International Workshop from Software Requirements to Architectures, Toronto, Canada (2001)

[Object Management Group, 1999a] Object Management Group: Fault Tolerant CORBA Specification, OMG document orbos/99-12-08 ed., OMG, Needham, Mass. (1999a)

[Object Management Group, 1999b] Object Management Group: The Common Object Request Broker: Architecture and Specification, 2.3 ed., Framingham, Mass. (1999b)

[Object Management Group, 2000] Object Management Group: The Common Object Request Broker: Architecture and Specification, 2.4 ed., Needham, Mass, OMG (2000)

[Oreizy et al., 1998] Oreizy, P., Medvidovic, N., Taylor, R., and D. Rosenblum, D.: Software Architecture and Component Technologies: Bridging the Gap. In Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA (1998)

[Othman et al., 2001a] Othman, O., O'Ryan, C., and Schmidt, D.C.: Strategies for CORBA Middleware-Based Load Balancing. IEEE Distributed Systems Online, 2(3) (2001a)

[Othman et al., 2001b] Othman, O., O'Ryan, C., and Schmidt, D.C.: Designing an Adaptive CORBA Load Balancing Service Using TAO. IEEE Distributed Systems Online, 2(4) (2001b)

[Parnas and Weiss, 1985] Parnas, D.L., and Weiss, D.: Active Design Reviews: Principles and Practices. In: Proceedings of the 18th International Conference on Software Engineering (1985)

[Parnas, 1972] Parnas, D.L.: On the Criteria to Be Used in Decomposing Systems into Modules, Communications of the Association of Computing Machinery, 15(12) (1972)1053-58

[Parnas, 1976] Parnas, D.L.: On the Design and Development of Program Families. IEEE Transactions on Software Engineering, 1 (1976) 1-9

[Parnas, 1979] Parnas, D.L.: Designing Software for Ease of Extension and Contraction, IEEE Transaction on Software Engineering, 5 (2) (1979)

[Parnas, 1994] Parnas, D.L.: Software Aging. In: I6th International Conference on Software Engineering, Sorento, Italy, ACM Press (1994)279-287

[Port et al., 2002] Port, D., Huang, L., and Boehm, B: Strategic Architectural Flexibility. In: 4th International Workshop on Economics-Driven Software Engineering Research (EDSER), (2002) 32-37

[Pree, 1994] Pree, W.: Design Patterns for Object-Oriented Software Development, Addison-Wesley, Reading, MA (1994)

[Rapanotti et al., 2004] Rapanotti, L., Hall, J., Jackson, M., and Nuseibeh, B.: Architecture Driven Problem Decomposition. In: Proceedings of 12th IEEE International Requirements Engineering Conference (RE'04), Kyoto, Japan, IEEE Computer Society Press (2004) 80-89

[Ross et al., 1996] Ross, S. A., Westfield, R.W., and Jaffe, J.: Corporate Finance (Fourth). Irwin, Chicago (1996)

[Saaty, L, 1980] Saaty, L.: The Analytical Hierarchy Process. New York: McGraw-Hill (1980)

[Schaller, 1999] Schaller, R.R.: Technology Roadmaps: Implications for Innovation, Strategy, and Policy, The institute of Public Policy, George Mason University Fairfax, VA (1999)

[Schmid, 1998] Schmid, H.A: Design Patterns to Construct the Hot Spots of a Manufacturing Framework. In: The Patterns Handbook: Techniques, Strategies and Applications, L. Rising. Cambridge University Press, Cambridge, UK (1998) 443-470

[Schmidt et al., 1998] Schmidt, D.C., Levine, D.L., and Mungee, S.: The Design and Performance of Real-Time Object Request Brokers, Computer Communication, 21(4) (1998) 294-324

[Schmidt et al., 2000] Schmidt D.C. et.: Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2, John Wiley & Sons, New York (2000)

[Schwartz and Trigeorgis, 2000] Schwartz, S., and Trigeorgis, L.: Real Options and Investment under Uncertainty: Classical Readings and Recent Contributions. MIT Press Cambridge, Massachusetts (2000)

[Shaw et al., 1995] Shaw, M., DeLine, R., Klein, D., Ross, T., and Young, D.: Abstractions for Software Architecture and Tools to Support them. IEEE Transactions on Software Engineering, 21(4) (1995) 314-335

[Shepperd et al., 1996] Shepperd, M., Schofield, C., Kitchenham, B.: Effort Estimation Using Analogy. In: Proceedings of the 18th international conference on Software Engineering, IEEE Computer Society, 170-178 (1996)

[Simon et al., 2001] Simon, F. Steinbru, F. and Lewerentz, C.: Metrics Based Refactoring. In: Proceeding of the European Conference on Software Maintenance and Reengineering (2001) 30-38

[Smith and Woodside, 1999] Smith, C., and Woodside, M.: System Performance Evaluation: Methodologies and Applications. CRC Press (1999)

[Smith, 1990] Smith, C.: Performance Engineering of Software Systems. Addison-Wesley, Reading, MA (1990)

[Stafford and Wolf, 2001] Stafford, J. A., and Wolf, A. W.: Architecture-Level Dependence Analysis for Software System. International Journal of Software Engineering and Knowledge Engineering, 11(4) (2001) 431-453

[Stafford et al., 1997] Stafford, J.A., Richardson, D.J., and Wolf, A.L: Chaining: A Software Architecture Dependence Analysis Technique. Technical Report CU-CS, Department of Computer Science, University of Colorado, Boulder, CO (1997) 845-97

[Stroulia and Leitch, 2003] Stroulia, E., and Leitch R.: Understanding the Economics of Refactoring. In: Proceedings of the Fifth ICSE Workshop on Economics-Driven Software Engineering Research (2003)

[Stultz, 1982] Stultz, R.: Options on the Minimum or the Maximum of Two Risky Assets: Analysis and Applications, Journal of Financial Economics, 10(1982)161–85

[Subramanian and Breslawski, 1993] Subramanian, G.H. and Breslawski, S.: Dimensionality Reduction in Software Development Effort Estimation. Journal of Systems and Software, vol. (21) (2) (1993) 187-196

[Sullivan et al., 2001] Sullivan, K.J., Griswold, W., Cai, Y., and Hallen, B.: The Structure and Value of Modularity in Software Design. In: the Proceedings of the ninth ESEC/FSE, Vienna, Austria (2001) 99-108

[Sullivan, 1996] Sullivan, K. J.: Software Design: The Options Approach. In: the Proceedings of the Second International Software Architecture Workshop. Joint Proceedings of the SIGSOFT '96 Workshops, San Francisco, CA (1996) 15–18

[Sullivan, 1997] Sullivan, K. J., Socha, J., and Marchukov, M.: Using Formal Methods to Reason about Architectural Standards. In: Proceedings of the 19th International Conference on Software Engineering, Boston, MA, ACM Press (1997) 503-513

[Sullivan, 1999] Sullivan, K. J.: Chalasani, P., Jha, S., and Sazawal, V.: Software Design as an Investment Activity: A Real Options Perspective. Real Options and Business Strategy: Applications to Decision-Making. In: Trigeorgis L. (ed.) Risk Books (1999) 215-260

[Sun MicroSystems, 2002] Inc Sun MicroSystems Inc: Enterprise JavaBeans Specification v2.1 (June 2002)

[Sun Microsystems] Sun Microsystems Inc.: Duke's bank application, http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html.

[Trigeorgis, 1995] Trigeorgis, L.: Real options in Capital Investment: Models, Strategies, and Applications. Praeger Westport, Connecticut London (1995)

[Valmari, 1991] Valmari, A.: A Stubborn Attack on State Explosion. In: E. M. Clarke and R. Kurshan(editors), Computer-Aided Verification 90. American Mathematical Society, Providence RI. Number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, (1991) 25–41

[van Lamsweerde, 2000] van Lamsweerde, A.: Requirements Engineering in the Year 00: A Research perspective. In: Proc. 22nd International Conference on Software Engineering, Limerick, Ireland (2000) ACM Press 5-19

[Vestal, 1996] Vestal, S.: MetaH Programmer's Manual, Version 1.09, Technical Report, Honeywell Technology Center (1996)

[Walkerden and Jeffery, 1999] Walkerden F. and Jeffery R: An Empirical Study of Analogy-based Software Effort Estimation. Empirical Software Engineering, vol. (4) (2), 135-158 (1999)

[Williams and Smith, 1998] Williams, L.G. and Smith, C.U.: Performance Evaluation of Software Architectures. In: Proceedings of the Workshop on Software and Performance (WOSP98), Santa Fe, NM (1998)

[Yau et al., 1978] Yau, S., Collofeloo J.S., and MacGregor T.: Ripple Effect Analysis of Software Maintenance. In: Proceedings of Compsac, IEEE Computer Society Press, Los Alamitos, CA (1978) 60-65